Jül-4339

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

A priori minimisation of algorithmic bottlenecks in the parallel tree code PEPC

H. Hübner

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

A priori minimisation of algorithmic bottlenecks in the parallel tree code PEPC

H. Hübner

Berichte des Forschungszentrums Jülich; 4339
ISSN 0944-2952
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)
Jül-4339

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL)
unter http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Dedicated to my beloved grandma
Inge Emma Wanda Dorothea Rust.
I will ever try to act according to your principles.
I will never forget you!

Abstract

The challenging problems arising from fast parallel N-body simulations became a driver for high performance computing. The Barnes-Hut tree code is an example in the class of fast summation algorithms, with a complexity of $\mathcal{O}(N \log(N))$, instead of $\mathcal{O}(N^2)$. The multi disciplinary code PEPC – the 'Pretty Efficient Parallel Coulomb solver' – is based on the Hashed-Oct-Tree scheme and is developed at Jülich Supercomputing Centre.

The pure bookkeeping overhead of the data-distributed tree construction decreases the performance and rapidly increases for large scales, as shown for JUGENE, an IBM Blue Gene/P architecture. For this reason, novel approaches will be established and applied, minimising integral bottlenecks. An axiomatic and provable optimisation of the parallel organisation structure, induced by a distributed memory machine, is introduced in detail. Reducing memory footprint and communication alike, the new concept intrinsically guides to a tight a-priori estimation of parallel data overhead. Moreover, the influence of the locality-preserving Hilbert-curve on the irregular communication structure, is studied.

Accordingly, the new method provides an immense upgrade for the particle number, making PEPC a more versatile tool for simulations in a multi disciplinary context.

Zusammenfassung

Die algorithmische Betrachtung und parallele Simulation des N-Körper Problems ist eine treibende Kraft und herausfordernde Aufgabe für das High Performance Computing. Ein schneller Summationsalgorithmus zur Reduktion der Komplexität ist der Barnes-Hut Tree Code mit $\mathcal{O}(N \log(N))$, anstatt der üblichen $\mathcal{O}(N^2)$ Wechselwirkungen. Im Jülich Supercomputing Centre wird auf der Grundlage des „Hashed-Oct-Tree“ Schemas der multidisziplinäre Code PEPC – der „Pretty Efficient Parallel Coulomb Solver“ – entwickelt.

Für eine sehr große Prozessoranzahl, wie zum Beispiel der des Superrechners JUGENE, einer IBM Blue Gene/P Architektur, wächst der parallele Mehraufwand zur Baumkonstruktion enorm und wirkt sich nachteilig auf das Skalierungsverhalten aus. In der vorliegenden Arbeit werden aus diesem Grund neue Verfahren zur Optimierung wesentlicher Engpässe hergeleitet und umgesetzt. Eine grundsätzliche und beweisbare Minimierung der Organisationsstruktur, die der Parallelität des Algorithmus geschuldet ist, wird vorgestellt. Als entscheidender Vorteil des neuen Ansatzes erweist sich eine präzise a-priori Abschätzung der parallelen Datenstruktur und damit verbunden, eine Reduktion des Speicherbedarfs, sowie des Kommunikationsaufwandes. Weiterhin wurde zur Verbesserung der Datenlokalität die Hilbert-Kurve implementiert und ihre Auswirkung auf die irreguläre Kommunikationsstruktur untersucht.

Durch die Optimierung des Speicherverbrauchs erlauben die Ergebnisse der neuen Methode eine immense Steigerung der Teilchenanzahl sowie den Einsatz des massiv parallelen Barnes-Hut Tree Code PEPC als noch flexibleres Tool hinsichtlich der Simulation in einem multidisziplinären Kontext.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

> "...while grouping together increasingly large groups of particles at increasingly large distances. This corresponds to the way humans interact with neighboring individuals, further villages and increasingly further and larger states and countries – driven by increasing cost and decreasing need to deal with more removed groups on an individual basis."
>
> Josh Barnes and Piet Hut, 1986

> "A resident of Lower-Wobbleton, Kent, England, is unlikely to untertake a trip to Oberfriedrichsheim, Bavaria, Germany, for a beer and to catch up on the local gossip."
>
> Paul Gibbon and Susanne Pfalzner, 1995

Since the advent of the Z3 [1], the first modern computer whose structure was independent of the treated problem, invented by Konrad Zuse [2] in 1941, there was an enourmous developement in the past seven decades. Exemplary for that is the increase of transistors on a chip described by Moore's law [3]. The era of petascale computing and supercomputers, where many chips are combined via highspeed network, has long since been entered. Next to the performance of the hardware, fast and efficient software is required to use the resources optimally. But even with many billions of operations per second the algorithmic complexity limits the evalution of some scientific problems in forseeable time. The simulation time can be much longer than the lifetime of our sun, longer then the lifetime of the scientist, naturally.

A classical system that consists of N particles interacting through long-range forces, called N-body problem, is a famous example in this class of problems and has interested physicists for centuries, because it is appearing in a wide range of applications and in many disciplines [4]. Each particle moves according to Newton's equations of motion, which can be calculated by various numerical integrators provided that the stimulating

force is known. In case of the N-body problem the force involves a fragment that is hard to compute. Since a long-range force (generally a $1/r^n$ potential) has an infinite scope, every particle-particle interaction has to be taken into account. This results in the algorithmic order $\mathcal{O}(N^2)$. For large particle numbers this naive direct algorithm is not feasible despite supercomputers.

In the 80s of the 20th century the Barnes-Hut tree algorithm [5], further referred as the BH tree code, was published reducing the order to $\mathcal{O}(N \log(N))$. The introductory quotes should describe the basic idea of this method expressed by the projection to human behaviour. Particles that are far away fullfilling a separation criterion [6, 7] are grouped to a pseudo-particle, which can be easily built using a tree structure. The larger the distance, the more often the condition of the criterion is fulfilled and forces are approximated. For the sober Lord the drunken Bavarians are far enough away, so they are lumped together and their blabber does not extend to England, whereas the tea time talk with like-minded noblemen reaches his ear. By this approximation, a controllable error is introduced, and only one interaction with the pseudo-particle is neccessary and the interaction with the individual particles is omitted. The effect of the pseudo-particle on a single particle is approximated by a tailor expansion up to a given order of the potential function. When the separation criterion fails, the particle-particle interaction is calculated just like the direct summation. Because the numeric time integrator introduces an error anyway, the error of the estimation can be hidden.

At the Jülich Supercomputing Centre (JSC) [8], an institute of the Forschungszentrum Jülich GmbH [9], the parallel tree code PEPC (Pretty Efficient Parallel Coulomb solver) [10, 11] was developed over the past few years. This highly portable code, based on the original Warren-Salmon-HOT scheme (Hashed Oct-Tree) [12, 13], challenges the porting of the sequential tree code to distributed memory machines. The basic kernel, which computes the long-range potential, is coated by a second order leap frog integrator and various front-ends. This are standing for multi disciplinary many body problems like complex plasma systems [14], gravitional [15] or coulombic problems and vortex fluids methods [16, 17, 18]. In addition other potentials differing from the standard $1/r$ potential are implemented. This makes PEPC a high scalable tool for various physical applications. Tuning the physics independent kernel that computes the pair potential, means tuning all front-end applications. Therefore in this thesis it is paid less attention to the various N-body systems, but on the optimisation of the tree code kernel.

In Chapter 2 the elementary, physical context and the basics of parallel programming are introduced. Furthermore, important parallel architectures, which were used to benchmark the implementation and are hosted at JSC, are described.

In Chapter 3, the program PEPC is described in detail. Linked to this is the precise description of the HOT-scheme. Special emphasis is placed on the possible bottlenecks, that arise by the complex porting to a distributed memory machine.

After the presentation of the program and its bottlenecks, novel approaches are implemented, discussed and statistically examined in the following chapters.

One the one hand in Chapter 4 space-filling curves that are adequate for the HOT-scheme are generally presented. Furthermore the Hilbert-curve [19, 20] is integrated in PEPC and compared with the previously installed Morton-curve [21, 20]. Thanks to its excellent locality preserving character [22] the Hilbert-curve promises an increase of performance of the tree code at various stages through the decrease of communication. The generation of Hilbert-derived keys is slower than the Morton-order. Here fast algorithms with the same complexity like the Morton-order mapping are considered and generalized on all possible patterns of the Hilbert-curve as well as novel space-filling curves.

On the other hand, Chapter 5 deals with the minimisation of the major bottleneck of the HOT-sheme, the combination of P local trees to a global data-distributed structure, that is required for the essential force summation. The keyword for this bottleneck is the branch node. This administrative unit, and the shortcoming of the current algorithm for the determination, is examined in detail. For a novel introduced concept, the multitude of advantages is explained. A tight a-priori upper bound for the local and global number of branches is constructed, and used to derive a novel and fast branching algorithm. Finally, the new concept is compared with the original version for extreme scales.

Chapter 2

The Basics

The N-body problem is the physical basis of this work. Introducing into the matter of simultaneous motion of N particles interacting through long-range forces, this chapter also embraces a classification of the BH tree code in the plurality of N-body solver. Since a parallel version of this algorithm is the main focus of this thesis, a short overview of parallel programming and the supercomputing resources at JSC is made.

2.1 The N-body Problem: Introduction and Solver

Fundamentally for classical, mechanical motion of any object is the second Newtonian axiom [23]. This means that the force is coming from the temporal change of the impulse $\vec{p}$:

$$\vec{F} = \frac{\partial \vec{p}}{\partial t}.$$

Dealing with N particles with mass $m_i \in \mathbb{R}_>$ and the position $\vec{r}_i = (x_i, y_i, z_i)^\mathsf{T} \in \mathbb{R}^3$. The impulse for every particle is defined as $\vec{p_i} = m_i \vec{v}_i$ where $\vec{v}_i$ is the velocity that is the change of the position by the time. In other words $\vec{v}_i = \partial \vec{r}_i / \partial t$. Overall this means for Equation 2.1 and the ith particle:

$$\vec{F}_i = m_i \frac{\partial \vec{v}_i}{\partial t} = m_i \frac{\partial^2 \vec{r}_i}{\partial t^2}. \tag{2.1}$$

Various forces can have an effect on every particle. The superposition principle says that all particular forces $\vec{F}_1, \ldots, \vec{F}_n$ accumulate to $\vec{F}$. Additionally it will be assumed that every particular force can be expressed by a potential function $\vec{F}_j = -\nabla V_j$. A potential function exists, if the work integral is independent of the path. In the end $\vec{F} = -\nabla V$. A combination of Equation 2.1 and the prior result delivers a second order ODE (Ordinary Differential Equation) for the ith particle. So for N particles a system of N ODEs results.

Provided that the force involves a long-range part with an infinite scope, this system is called the N-body problem:

$$m_i \frac{\partial^2 \vec{r}_i}{\partial t^2} = -\nabla_i V, \quad i = 1, \ldots, N. \tag{2.2}$$

We concentrate on potentials V, that can be splitted in two independent parts:

$$V = V_{pair} + V_{ex}.$$

The external part V_{ex} results from external forces. The part V_{pair} comes from the force the particles exert on each other. The pair potential V_{pair} can again be divided into:

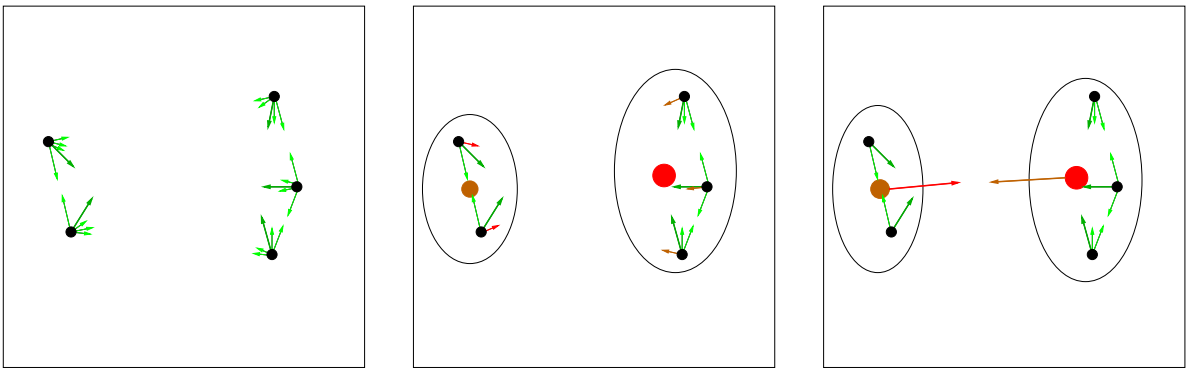
$$V_{pair} = V_{short} + V_{long}.$$

It is distinguished between a short-range potential V_{short} and the earlier mentioned long-range potential V_{long}. The short-range potential is quickly decaying and at a certain distance between the interacting particles, the force is identically zero. Contrary to that is the long-range potential, that makes a contribution to the force for every distance. To calculate the trajectories of many particles numerical methods are used, because for more than two particles a general, analytical solution does not exist. For the equation of motion a typical one step integration scheme, like the first order Euler or the second order Leap-Frog (Stoermer-Verlet scheme) can be used [24]. For the second scheme it is not neccessary to transform the differential equation to a first order system. Higher order Runge-Kutta schemes need many evaluations of the the right hand side which can be very expensive in this case and are therefore neglected. In the future the use of multi step schemes like the BDF (Backward Differentiation Formulas) can be tested. However, it is in dispute whether a higher accuracy is needed for tree codes and momentums are conservated. But for all integration schemes it is necessary to know the potential V.

Getting V_{ex} represents no challenge, because it is independent of the number and position of the particles and can be calculated seperately for each particle. So a computational effort of $\mathcal{O}(N)$ results. As well as the external field the short-range potential V_{short} can be calculated with $\mathcal{O}(N)$, because the potential decays rapidly and for each particle only a small set of nearest neighbours has to be taken into account. To evaluate V_{long} each of the N particles must interact with every other particle $(N-1)$. Due to Newton's third law (actio=reactio) [23] the number of operations is indeed halved to $N(N-1)/2$ but still of the order $\mathcal{O}(N^2)$. In Figure 2.1(a), the partial (light green) and the overall force (dark green) is shown. Even in the petaflop era the long-range potential cannot be calculated directly for large particle ensembles. Therefore, faster N-body solver, like the BH tree code, are needed. In the class of solvers it is distinguished between grid-free and grid-based methods.

A regular grid of size $M \times M$ can be layed over the region which includes all N particles. Then the particle properties are used to create a source density, which is interpolated at the grid points. Colloquially, the properties of the particles are mapped onto the grid

points. This step requires an operation count like $\mathcal{O}(N)$. Afterwards the underlying physical equation is solved only for the grid points. Then the force is back-transformed from the grid points on the particles. The algorithmic complexity of the so called PIC-method (Particle In Cell) [25] depends on the underlying equation. E.g. handling a static evaluation of the pair potential, then an elliptical partial differential equation (i.e. a Poisson equation, if source term exists) must be solved. The grid provides a discretisation of this equation like the finite-difference method. It can be solved with the help of a fast Poisson-solver [26], that uses FFT (Fast Fourier Transform) [27], with the algorithmic order of $\mathcal{O}(M \log(M))$. Hence an overall algorithmic complexity of $\mathcal{O}(N + M \log(M))$ is achieved for this example. Since in practice the number of particles N is much bigger than M ($M \ll N$) a scaling like $\mathcal{O}(N)$ can be observed.



(a) Direct summation (b) BH tree code (c) FMM

Figure 2.1: Comparison of different grid-free N-body solvers regarding to the calculation of the long-range potential on the basis of minimal example with $N = 5$.

The PIC-method has problems with complex geometries and strongly non uniform particle distributions. It can be reformed by advanced techniques like adaptive grids. For open boundary problems, this method imposes partly artificial boundaries. For other simulations, that adhere to problems with boundary conditions, like TOKAMAK [28] simulations, the boundary imposing character of the PIC-method is very useful. There are other derivates of the PIC-code, like the P^3M-method (Particle-Particle–Particle-Mesh) [29], but a further examination would go too far in this thesis.

In contrast to the PIC-method the direct summation is a grid-free method. The problem of an infeasible run time – for large particle ensembles – and the disadvantages of grid-based methods – not only for open boundary problems – are overcome by two multipole based techniques namely the BH tree code (Barnes-Hut tree code) [5] and the FMM (Fast Multipole Method) [30], which were introduced in the mid-1980s. Their algorithmic scaling is of the order $\mathcal{O}(N \log(N))$, respectively $\mathcal{O}(N)$.

Both algorithms have in common that distant particles, with a small contribution to the force of an individual particle, are clustered to a pseudo-particle. To obtain the properties of the pseudo-particle, e.g. the center of charge, the averaged particle properties are used.

The BH tree code does not need a grid in the usual sense, but a clever division of the arbitrary space that admits a relationship between particles and its neighbours to be obtained. In practice mainly 3D or 2D problems are considered. Firstly, it is started with a cube (3D) or a plane in a squared format (2D) that contains all particles. Then this region is subdivided into 8 (3D) or 4 (2D) sub cubes simply by halving every space direction. If one sub cube does not include any particle, then it is discarded. If it includes exactly one particle, then the subdivision is dropped out. If more particles are included, then the process is continued until only sub cubes with exactly one particle are remaining. All occupied boxes can be cleverly saved in a tree structure (became nodes). The tree include nodes with either one (leaf) or more particles (twig). Every particle traverses the tree and for every node it will be decided whether the pseudo-particle, that is built out of all particles this box includes, can be used or if the box must be more refined. Therefore a separation criterion will be needed.

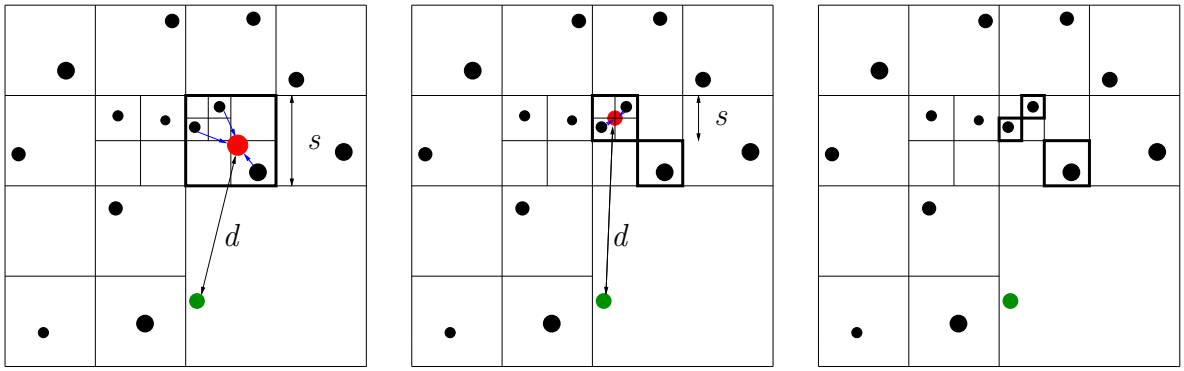


Figure 2.2: Space division and separation criterion in the Barnes-Hut tree code applied to a 2D particle arrangement. The pseudo-particle is dyed red and blue arrows mark particles which contribute their properties. Only the interaction of the green particle with the box is considered. The box which includes the pseudo-particle is highlighted. Three steps of the tree walk are displayed. (a) 3 particles are grouped. This pseudo-particle will be accepted for $\theta > 0.54$. (b) If the MAC fails the box will be further refined. The two particles in the upper left box are grouped to a pseudo-particle. With the lower right particle it will be interacted directly. This pseudo-particle will be accepted as interaction partner for $\theta > 0.23$. (c) If the MAC fails again, then only leaves remain. A direct interaction with all leaves results.

A geometrical MAC (Multipole Acceptance Criterion) is the Barnes-Hut MAC in Equation 2.3. This computes the quotient between box length of the pseudo-particle s and the distance d between the particle and the center of charge of the pseudo-particle. If and only if this quotient is lower or equal to a control parameter $\theta \in \mathbb{R}_{\geq}$ the pseudo-particle with the multipole expansion can be used:

$$\frac{s}{d} \leq \theta. \tag{2.3}$$

For $\theta \approx 0$ the quotient has to be very small to accept the pseudo-particle. Consequently, it is used in case of a very small box and a very large distance. Thus small values of θ tend to use the particle-particle interaction, because almost every pseudo-particle does not fulfill the separation criterion. In contrast, $\theta \to \infty$ accepts every pseudo-particle. The user must find a compromise between saved interactions and the accuracy of the approximation. In practice $0.1 \leq \theta \leq 1.0$ seems to be a good decision. In Figure 2.2 the intrinsic space division in the BH tree code is shown. Furthermore a single green particle is taken to demonstrate the application of the BH MAC for one interaction with the highlighted box. There are plenty of other MACs with specific advantages for different applications [6, 7].

In Figure 2.1(b) the calculation of the forces in the BH tree code is shown. In this example, both left and the three right particles were summarized to a brown respectively a red pseudo-particle. In relation to the direct method some particle-particle interaction were saved. On the left hand side two interactions were saved each, because the individual interaction is cut out by the red pseudo-particle. On the right hand side one interaction is saved each, because instead of both left particles, only the brown pseudo-particle is taken into account. The other interactions correspond to the direct summation. Since the particles are so close, the approximation through a pseudo-particle cannot be applied [4]. A variant of the tree code is the Warren-Salmon-HOT scheme (Hashed Oct-Tree) [12]. This scheme is used in PEPC and is described in detail in Chapter 3.

The FMM is a refinement of the BH tree code. This method makes use of the fact, that a multipole expansion of infinite order contains the information about the entire particle ensemble. It has an algorithmic order of $\mathcal{O}(N)$ and was awarded as being one of the top 10 algorithms of the 20th century [31]. Whereas the BH tree code produces particle-particle and particle–pseudo-particle interactions, the FMM interacts directly for near particles and between pseudo-particle–pseudo-particle for far distances. Again, the algorithmic order is reduced from $\mathcal{O}(N \log(N))$ to $\mathcal{O}(N)$. In Figure 2.1(c) the elemental force summation of this approach is shown. The particle–pseudo-particle interaction in comparison to the BH tree code in Figure 2.1(b) is missing [30]. In Table 2.1 all presented N-body solvers are listed in chronologic order.

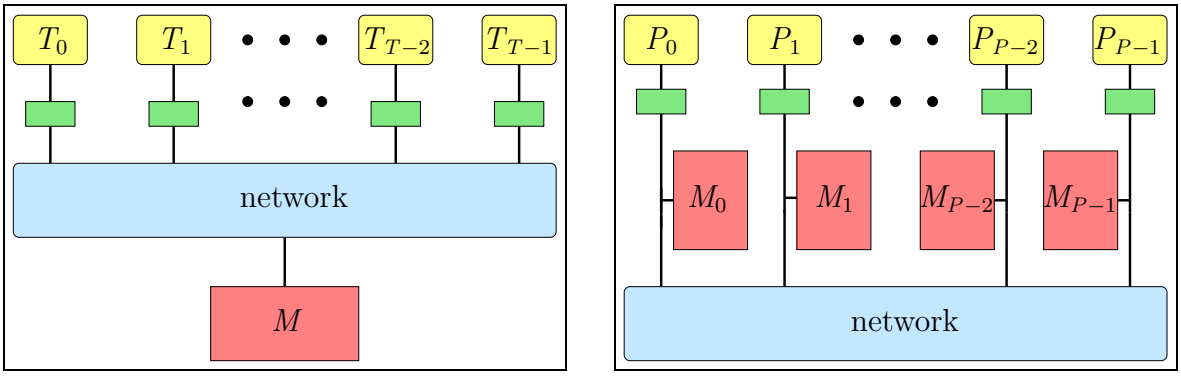
Solver	Complexity	Year
Direct Summation	$\mathcal{O}(N^2)$	1820
Particle-In-Cell (PIC)	$\mathcal{O}(N)$	1955 [32]
Barnes-Hut-tree code (BH tree code)	$\mathcal{O}(N \log N)$	1986 [5]
Fast Multipole Method (FMM)	$\mathcal{O}(N)$	1987 [30]

Table 2.1: A brief summary of important N-body techniques in chronologic order. Adapted from [33].

2.2 Parallel Programming: Preface and Characteristics

First a motivation for massively parallel architectures is given. Newer processor generations tend to include multiple cores. This run at an acceptable clock rate, being a compromise of clock speed, heat generation and power consumption, and share a common memory. On every core an individual thread can be programmed, resulting in T parallel units. Thence, T threads can perform a parallel work flow, and at a maximum with a T times faster performance. Today in the desktop and laptop segment usually quad core or oct core processors are commercially available. The most famous programming interfaces for so called shared memory architectures are libraries like OMP (Open Multi-Processing) [34] or PThreads (POSIX Threads) [35]. In contrast to OMP, where mainly loops are parallelized, PThreads allows each thread to perform different exercises, regardless of a loop. In Figure 2.3(a) a shared memory system is shown.

A different idea is to combine P single core or multi core processors via highspeed network into a massively parallel supercomputer. In addition other assemblies, e.g. a union of many vector units [36] or a cluster of GPUs (Graphic Processing Units) [37] exist. Another possibility, currently under research and developement, is to use GPUs as accelerators for standard processors to speedup especially costly program parts, which can be easily distributed onto millions of threads. In this thesis only MPP (Massively Parallel Processing) systems with an arrangement of homogeneous processors are considered. The distributed memory accumulates to $P \cdot M$. Such a system is called a distributed memory machine, as illustrated in Figure 2.3(b). Each of the P participants is called task. Simply considered the P-fold problem size can be calculated, but additional memory is needed by virtue of the parallel execution. In this case, tasks cannot directly access the memory of another task. Therefore, non-local data must be communicated over the highspeed network. Mainly use of MPI (Message Passing Interface) [38] is made, whereas other more specialized libraries exist [39]. MPI defines a standard, that is implemented by



(a) shared memory (b) distributed memory

Figure 2.3: Comparison between shared and distributed memory machines. Threads are marked with T. The memory bus of every Thread is reduced to a black line. Tasks are marked with P. Memory is abbreviated with M. The green boxes are caches.

various manufactorers and partly optimized for a target machine. MPI-functions are classified into point-to-point or collective and blocking or non-blocking operations. In the case of point-to-point operations just two partners are communicating, whereupon all parallel tasks of a specific communicator are involved in collectives. If an operation is blocking, then the program flow stops at this point until the communication is finished. In the case of a non-blocking operation, the communication will be handed to the network card, if available, and the program flow continues.

If the architecture is an association of many multi core processors, then it is possible to use both programming paradigms simultaneously. E.g. one task runs per processor plus multiple threads that speedup the local calculation. MPI-OMP as well as joined MPI-PThreads versions are possible. Incidentally, shared and distributed memory programming regards to MIMD (Multiple Data Multiple Instruction Stream) [40] architectures.

For the performance analysis of parallel implementations various metrics exist, which are described in the following. Furthermore, two widely known laws will be presented. Let P be the number of parallel tasks, threads or the overall number of parallel units, further named as tasks. Crucial for the parallel performance is the decomposition of the problem onto the tasks as well as the overhead caused by parallelism including the communication and dead time of a task waiting for another one, called idling.

Each global operation indicates a synchronisation of the tasks. The overall delay of this barrier is dominated by the slowest tasks, usually the tasks with the largest portion of work. Therefore, all other tasks are waiting for slowest task and possible computing time is wasted. To guarantee an efficent parallel execution it is neccessary to share approximately similar portions of work. This is called load-balancing, which extremely increases the

performance. In Figure 2.4 an imbalanced and balanced parallel program with two global operations is shown. Whereas in Figure 2.4(a) time is wasted, marked with red infill, in Figure 2.4(b) the work is homogeneously distributed. As a consequence, the reduction of idling times leads to a shorter overall run time.

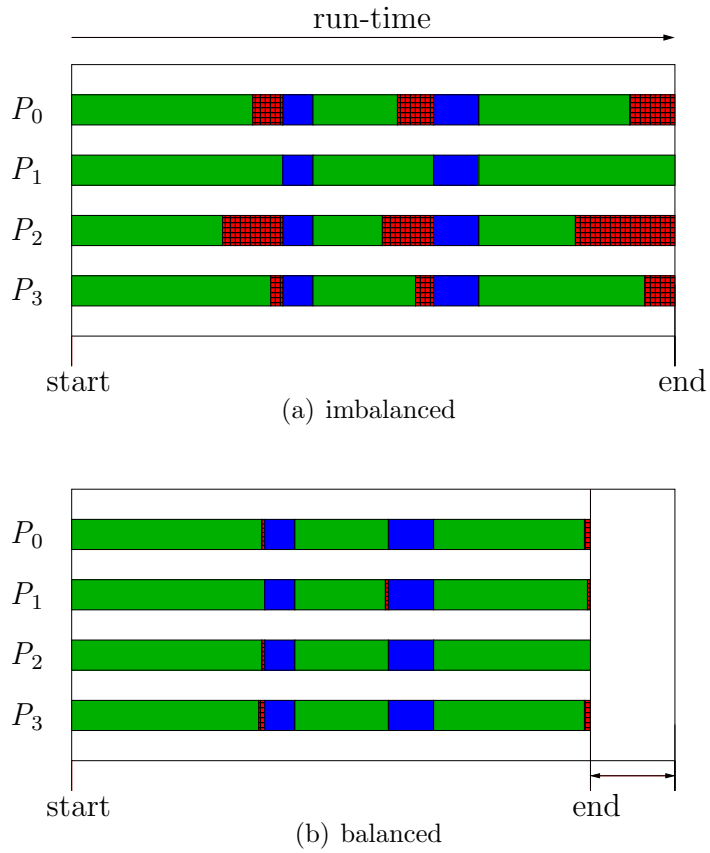


Figure 2.4: Demonstration of load-balancing for 4 tasks. Green infills are working tasks, whereas a red fill describes latencies. Blue colored intervals denote global operations or barriers.

The timings refer to the run time, that is the real elapsed time. Additionally there is the system time, that describes the time the processor needs for computation. If this time would be used for benchmarking, then communication, as a characteristic of the network, and idling as a measure for load-balancing, would not taken into account, which makes no sense.

A first metric to rate parallel performance is the speedup $S(P) \equiv S_P$. That depicts the acceleration of the program obtained by parallel execution. In this value the serial and parallel runtime $T(1) \equiv T_S$ respectively $T(P) \equiv T_P$ are composed. Also other normalising factors can be used for $T(1)$ instead of the fastest serial algorithm, hence:

$$S_P : \mathbb{R}^2_> \ni (T_P, T_S) \to S_P(T_P, T_S) := \frac{T_S}{T_P} \in \mathbb{R}_>. \tag{2.4}$$

Consequently the speedup, as a function of the number of tasks P, is displayed in a plot were P is the abscissa and S_P is the ordinate. The parallel overhead T_O is defined as:

$$T_O : \mathbb{N} \times \mathbb{R}^2_> \ni (P, T_P, T_S) \to T_O := PT_P - T_S \in \mathbb{R}_{\geq}. \tag{2.5}$$

The highest speedup can be achieved when parallel overhead does not exist. Hence the first angle bisector arises and $\sup_{p \geq 2}(S_P) = P$, which signalises the ideal linear speedup. In special situations a super linear speedup can be observed, that means $S_P > P$. This can be explained as follows: with a constant problem size and increasing number of tasks the amount of data for a single task is decreasing. Finally, at a certain number all data fit in the cache. Access to this is much faster than accessing the main memory, which is well known as a bottleneck. Therefore, the time for a communication with the main memory is vastly reduced, which has a positive effect on the overall runtime and the speedup, naturally. So this can be even greater than P. For scalability a distinction is made between strong-scaling, where the problem size remains constant, and weak-scaling, that describes the speedup curve for a fixed problem size per task. A metric for the exploitation of parallel resources emerging naturally by the speedup is the efficiency $E(P) \equiv E_P$:

$$E_P : \mathbb{R}_> \ni S_P \to E_P(S_P) := \frac{S_P}{P} \in]0, 1]. \tag{2.6}$$

At the maximum efficiency all resources were utilized, whereas in the minimum case it is pointless to investigate in paralellism, since one task brings the same effect and the resources are wasted.

So far the speedup respectively the efficiency were calculated by the sequential and parallel timings. In the mid-sixties of the 20th century a far sighted law was established by Gene Amdahl. This law rates the speedup regarding to the sequential, the not parallelisable part α and its counterpart $1-\alpha$. After Amdahl had written the correlations between both parts only in prose [41], it was transfered in a formula since known as Amdahl's law using an α-dependent formulations for the parallel and serial run time:

$$[0, 1] \ni \alpha \to S_P(\alpha) = \frac{\alpha + (1 - \alpha)}{\alpha + \frac{1-\alpha}{P}} = \frac{1}{\alpha + \frac{1-\alpha}{P}} \in [1, P]. \tag{2.7}$$

If one takes more and more tasks, the parallelisable part converges towards a time of 0. The non parallelisable part reamins constant. Finally, this part dominates the time for large scales and can be observed in Figure 2.5(a). Amdahl's law refers to the strong-scaling, because an increasing problem size can conceal the non parallelisable part and makes the law weak. If a percentage of the run time is directly proportional to the number of tasks P, that will be called P-dependent, it is even worse. Hence, the run time increases with the number of tasks for this part. E.g. if some data increases with at least $\mathcal{O}(\log(P))$ and this data must be exchanged. In this case, a much earlier saturation of the speedup can be observed, that is shown in Figure 2.5(b). There, the time has a minimum at 64.

Finally, for approximately 10000 tasks the parallel version is slower than the serial (not shown in Figure 2.5).

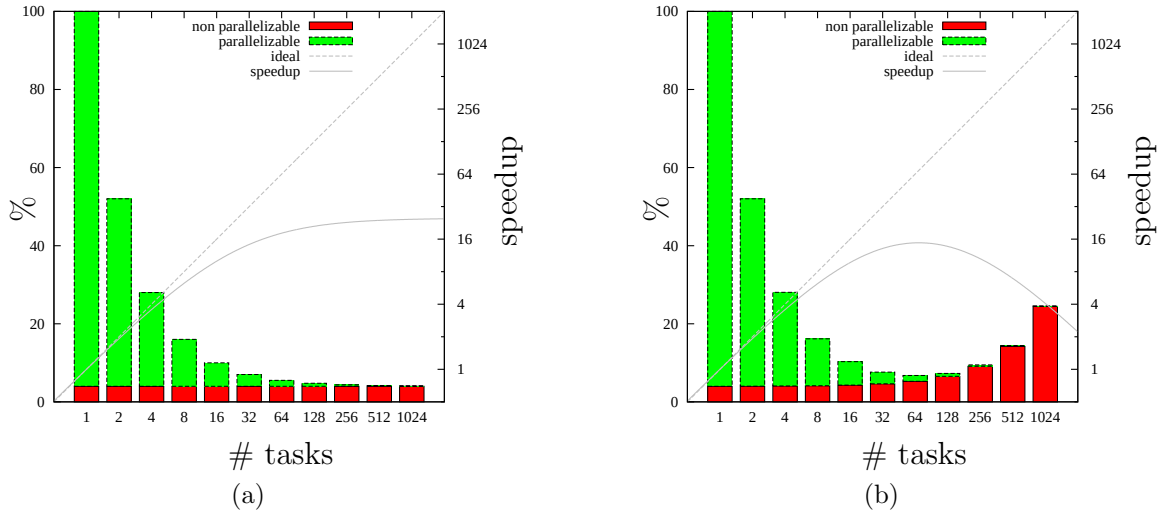


Figure 2.5: Impacts of Amdahls's law. Speedup for a constant non parallelisable and a P-dependent part. (a) constant non paralellisable part: $\alpha = 0.04$ (b) P-dependent part that grows according to $0.02 \cdot P + \alpha - 0.02$.

John L. Gustafson and his colleague Edwin H. Barsis were unsatisfied with Amdahl's law, because they have observed magnitudes larger speedups. Amdahl's law suggests, that the parallelisable part is independent of P. The handable amount of work in parallel varies linearly with P, at least for a distributed memory machines with $P \cdot M$ main memory. Therefore, the problem size can scale linearly with the number of tasks and the parallelisable part depends on P. Thus the run time is constant, but a much larger problem is calculated. This results in Gustafson's Law (also known as Gustafson-Barsis' law) [42], that motivates massively parallel application:

$$[0,1] \ni \alpha \rightarrow S_P(\alpha) = P - \alpha(P-1) = (1-\alpha)P + \alpha \in \ [1,P]. \tag{2.8}$$

The Equation 2.8 obeys a linear equation with slope $(1-\alpha)$ and constant term α. If the problem allows a flexible adjustment of the size, then a saturation of the speedup can be circumvented. This course of action is called scaled-scaling. For $\alpha = 0$ the linear speedup arises and for $\alpha = 1$ no speedup can be obtained. In Figure 2.6 various speedups are demonstrated depending on the chosen non paralelisable part α.

Nevertheless, also criticism for Gustafson's law exists. This law cannot directly used to judge the scaling property of a program with a P-dependent variable. It can be proved, that Amdahl's and Gustafson's law are equivalent [43]. A single effect is formulated in a different way: Amdahl's law normalizes the serial run time to 1, wheareas Gustafson

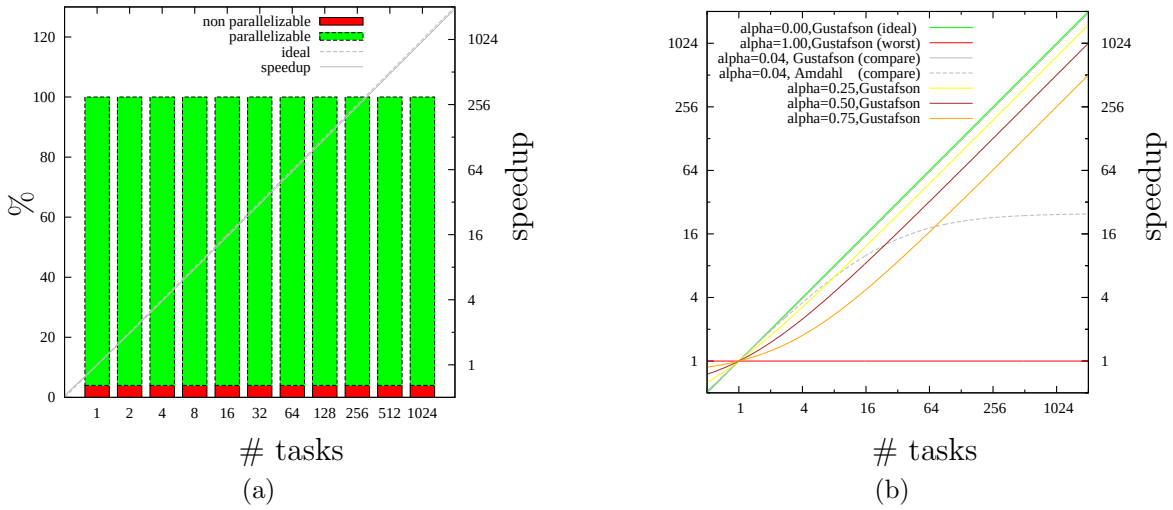


Figure 2.6: Visualisation of Gusatafson's law. A case study for different non parallelisable parts is shown. (a) $\alpha = 0.04$: The run time remains constant, but a multiple of the original problem size was calculated. The achieved speedup is near the ideal one and is not particularly easy to see. (b) The speedup according to Gustafson's law is plotted for various values of α. In case of $\alpha = 0.04$ both laws are compared. The curves do not even correspond to a linear equation (double logarithmic axis).

has a standardised parallel part and a varying sequential part. Anyhow, both laws are widely known in the HPC comunity and regarding to their date of publication both laws are milestones.

2.3 High Performance Computing Resources at JSC

To classify the resources of the JSC in a worldwide scale, firstly two crucial lists will be introduced. The TOP500 [44] is a list of the 500 most powerful computer systems, that will be compiled twice a year in June and November. The computers are ranked by their performance on the LINPACK (Linear Algebra Package) benchmark, that is measured in Flops (Floating Point Operations Per Second) and are listed in descending order. Additionally, some other parameters are listed. The maximal number of MPI-tasks is given in the column "Cores". With "R_{Peak}" the theoretical peak-peformance and with "R_{max}" the achieved performance of the LINPACK-benchmark is specified. Furthermore, the energy demand in kilo Watt is listed in the column "Power". The problem size for achieving the maximal and the half of the maximal performance, "N_{max}" respectively "$N_{1/2}$", are tabulated. The essential parameter in the GREEN500 [45], where the 500 most economical systems are listed, is the rate of MFlops per Watt.

The JSC hosts various supercomputers and is one of the leading HPC institutions in Europe. This institute follows a two tracked concept: a highly scalable and a general purpose cluster. The two most important systems, JUROPA and JUGENE, are summarized in Table 2.2 and shown in Figure 2.7 and 2.8.

Figure 2.7: In the foreground the grey JUROPA-JSC can be seen. In the background the black HPCFF part is arranged [9].

JUROPA (Juelich Research On Petaflop Architectures) [46] is a joint-venture of JUROPA-JSC and HPCFF (High Performance Computing For Fusion). The second part is dedicated to the nuclear fusion community. The whole systems consists of $2208 + 1080$ compute nodes. Every node is built of two Intel XeonX5570@2.93GHz quad core processors with Nehalem's 45nm micro-architecture and 24 GB of DDR3 main memory. The nodes are combined by a fat-tree topology. Maximally $17664 + 8640$ cores can theoretically perform $207 + 101$ TFlops at peak. On every node a SLES11 (Suse Linux Enterprise Server 11) operating system is running. This can lead to OS-jitter [47], since the parallel program is scheduled with jobs of the operating system. Demonstratively, various runs of the same program with the same input generate different run times. The system is currently placed at 23rd position worldwide and slightly far behind at 207th position of the GREEN500.

Figure 2.8: Front-view of the Jülich Blue Gene/P solution JUGENE. The 72 racks are distributed among nine units that contain 8 racks each. [9]

JUGENE (Juelich Blugene) [48], a highly-scalable Blue Gene/P Solution , completes the dual concept at JSC. The system contains 72 racks, containing in turn 32 node-cards. One node-card shelters again 32 compute-cards with 4 PowerPC450@0.85GHz and

2 GB DDR2 main-memory. Accordingly the whole system allows a maximal number of 294912 ($72 \cdot 32 \cdot 32 \cdot 4$) MPI-tasks to perform in parallel. The system JUGENE heads all other worldwide systems in the matter of the number of cores (November 2010). The main network is a 3D torus. Additionally, a network for collective operations like broadcast and reduction operations is installed. A network for global barriers and interrupts with a low latency completes the system. Europe's second fastest supercomputer is ranked at 9th position in the TOP500 with a theoretical peak performance of 1 PFlops and a LINPACK performance of 0.826 PFlops. The economical assembly of the Blue Gene/P is underlined by the 29th place in the GREEN500.

Computer/Year Vendor	Rank	Cores	R_{max}	R_{Peak}	Power	Rank	MFlops/W
JUGENE - Blue Gene/P Solution / 2009 IBM	9	294912	825.50	1002.70	2268	29	363.98
JUROPA - Sun Constellation, NovaScale R422-E2, Intel Xeon X5570, 2.93 GHz, Sun M9/ Mellanox QDR Infiniband/ Partec Parastation/ 2009 Bull SA	23	26304	274.80	308.28	1549	207	177.40

Table 2.2: Snippets from the TOP500 [44] and GREEN500 [45] lists. A summary about fundamental parameters of the two most important systems at JSC is given. The green columns represent the GREEN500 values.

Mainly JUGENE is used to benchmark the implementations of this work, which exhibits diverse reasons. Firstly, it is a goal to push parallel tree codes to extreme scales and this systems provides the maximal number of tasks worldwide. Secondly, this architecture does only run a vanilla CNK (Compute Node Kernel). An SLES9 operating system for the compilation is available on the front-end nodes. Hence, OS-jitter is imperceptible and a highly comparability of the results is guaranteed.

Chapter 3

PEPC – a State of the Art Parallel BH Tree Code

Over the past few years at JSC, the parallel tree code PEPC (Pretty Effcient Parallel Coulomb-solver) was developed by a team of physicists, mathematicians and software developers. This highly portable code was originally designed for mesh-free modeling of complex plasma systems [49, 50, 51]. Thanks to its extendable assembly, the strict separation of front-end applications and the kernel, it has since been greatly extended to cover gravitational problems [52], smoothed particle hydrodynamics [53], fusion plasmas [14] and vortex particle methods for fluid dynamics [18]. The multi disciplinary character is highlighted in Figure 3.1.

Besides other attemps to parallelise the BH tree code, e.g. virtual shared memory versions [54] or distributed memory schemes with a geometrical domain decomposition [55, 56], the trailblazing HOT-scheme [12, 13] eclipses other schemes in the matter of success. Pepc is based on this scheme and uses a fixed multipole expansion up to quadrupole order, delivering a high accuracy.

Mainly this chapter is dedicated to mark the bottlenecks of the parallel HOT-scheme and the associated program Pepc, regarding memory footprint and communication overhead. Scaling analysis up to the entire JUGENE system are presented in Section 3.2. For a better understanding the HOT-scheme will be explained in detail in Section 3.1.

Using the initial particle distribution for the first time step t_1, at a later time t_i the particle distribution of the previous time step $t_{i-1} = t_i - \Delta t$ is taken. The dynamic comes from the particle pusher in the front-end, where Equation 2.2 is solved with a leap-frog scheme. Usually the timestep Δt is very small, results in a minimal modification of the particle position $\vec{r}_i$. Nevertheless, the data strucure is built afresh each timestep, since spatial meanderings must be respected.

For benchmarking of novel approaches, two distinct yet typical particle setups were selected using particle numbers N of $0.125 \cdot 10^6$, $8 \cdot 10^6$ and $2048 \cdot 10^6$. Both setups describe an overall two-component plasma of $n_e = n_i = N/2$ electrons and ions, being differently distributed in the unit cube. On the one hand, the box is homogeneously filled.

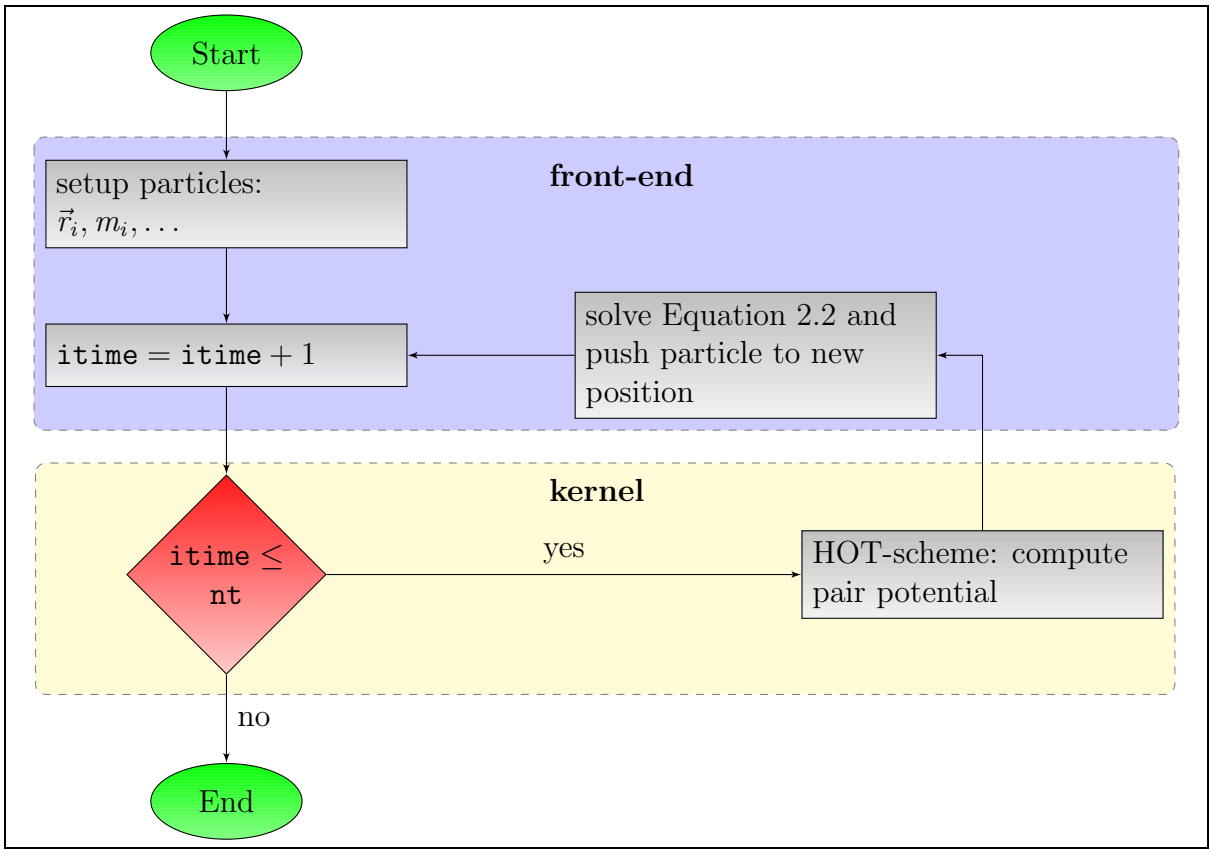
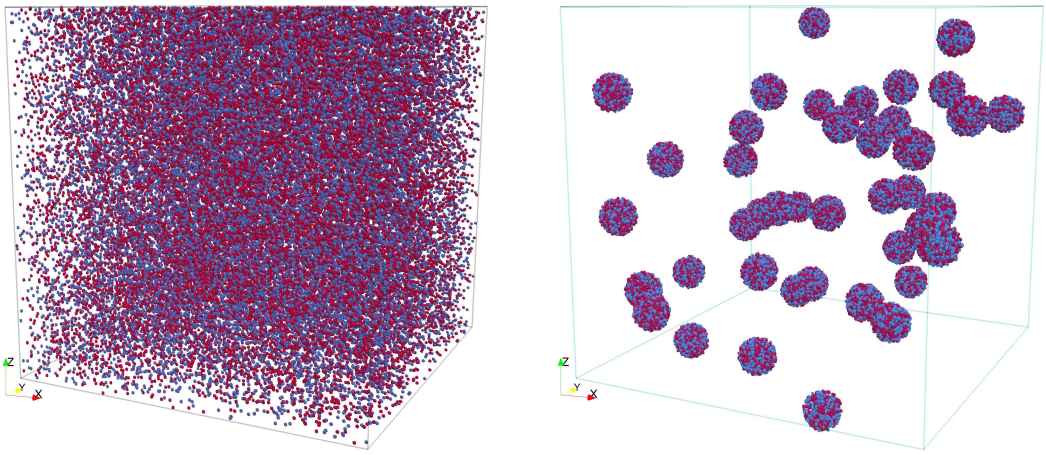


Figure 3.1: Multi disciplinary assembly of the parallel tree code PEPC. The blue area displays the physics dependent front-end. Here the particles are setup regarding to the underlying physical problem. The value `nt` denotes the number of timesteps. With a fixed timestep Δt, the current time t_i is calculated by `itime` $\cdot \Delta t$. The value `itime` starts with 0. The light-yellow area shows the physics independent kernel. For each timestep, the longe range pair potential is calculated via the HOT-scheme and then the particles are pushed to their new position.

On the other hand, a number of 42 randomly distributed and randomly filled spheres act as a placeholder for the inhomogeneous case. Every sphere contains averagely $N/42$ particles. In Figure 3.2, both, the homogeneous and inhomogeneous setups, are shown for 10000 particles in 3D. PEPC is a tree code primarily designed for 3D problems, but also handles 2D planes easily. In the following examples, such planes are used to improve geometric clearness. In this case, the data structure is a Hashed-Quad-tree.

3.1 The Parallel HOT-scheme

The idea of the BH tree code is to replace direct distant particle-particle interactions by the interaction with a pseudo-particle, that includes many particles and saves a manifold of operations. In order to keep the bookkeeping low, a tree structure is well-suited to



(a) homogeneous (b) inhomogeneous

Figure 3.2: Different initial distributions of 10000 particles for 3D.

check the separation criterion. The reinvent of the HOT-scheme regarding to the tree is the scrapping of the idea of memory pointers in favour of a set of universal binary keys to represent particle positions. These binary keys are intelligently chosen to determine tree properties with simple bit operations. The first choice for the generation of such keys are self-similar space-filling-curves, that is outlined in Chapter 4. The introduced disadvantage of a inconceivable large number of possible keys ($\approx 10^{19}$ for 64-bit integer) and the lack of satisfactory memory space is overcome by a hash table. In this table, the data structure is stored.

A quick overwiev of the flow of the HOT-scheme is outlined in the following and is displayed in Figure 3.3. After the decompositon of the particles among the tasks (`domain decomposition`), the local data structure is calculated. Since every task posesses only a subset of the entire particles, only a local tree can be built. In Figure 3.3, this step is named `local tree`. This local data structure is insufficient for the essential force calculation, where a global tree is neccessary. Therefore all local data structures are combined to a single data-distributed global tree. For the combination, the concept of branch nodes will be introduced. This set of keys subsumes the minimum number of complete nodes covering the entire local domain of each task.

After the branch information is broadcast across all tasks in the step `exchange`, a single virtual global tree can be built. Here, the branch nodes are an entry point for non-local information. Since nodes above the branch level may be incomplete, in the next step, they are built afresh subsequently from branch level to root. Then, a straighforward matter is the assigment of pseudo-particle properties to each node. Both, steps are combined to `global tree` in Figure 3.3.

Once the global structure is in place, every particle performs a traversal to build an interaction list. Non-local particle information must be requested from other tasks (with

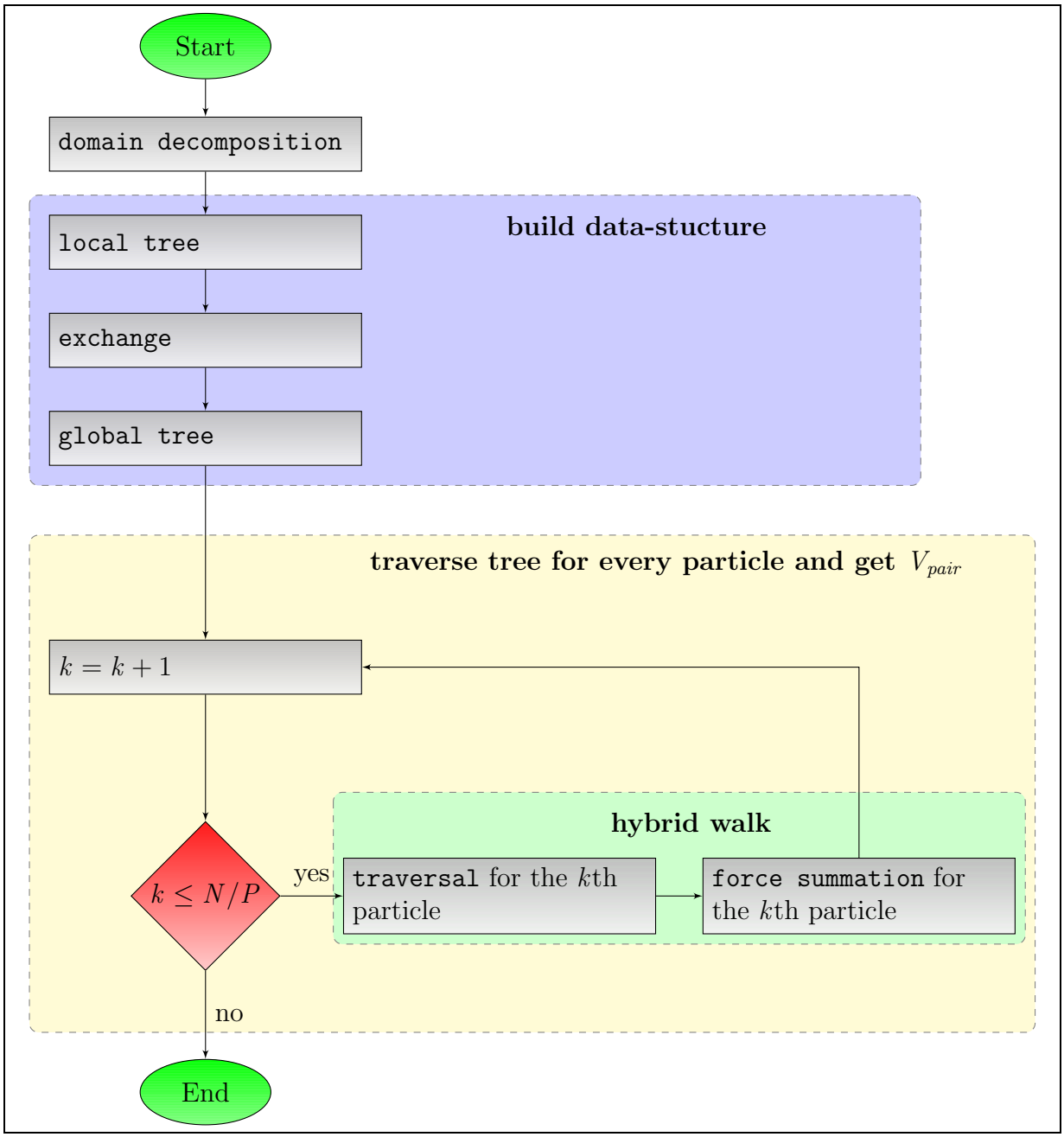


Figure 3.3: Flow of the HOT-scheme. The different phases are highlighted. It will be seperated between the construction of the data structure and the tree traversal for every particle. The hybrid version of the code is shown, where both `traversal` and the `force summation` are combined and communication to other tasks is performed by a parallel running thread.

the knowledge of all branch nodes), resulting in a complex communication scheme. After the interaction list is known, the force can be calculated for every particle. With the knowledge of the long-range pair potential V_{long}, the underlying ODE can be solved and the particle is pushed to its new position, but this part remains to the front-end. Then, the next timestep can be simulated in the same way.

3.1.1 Domain Decomposition

For the `domain decomposition`, only the particle positions $\vec{r}_i$ are required. The aim of this step is to distribute the particles among the tasks with respect to data locality and load-balancing.

For a given particle distribution, the simulation box shown in Figure 3.4 is determined as the smallest cube (edge length is denoted by L) containing all particles. Every task determines the local extrema and a reduce operation across all tasks brings the global minima and maxima and L can be derived.

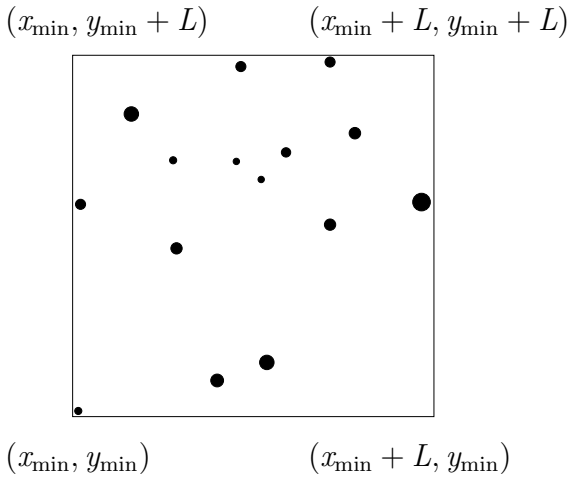
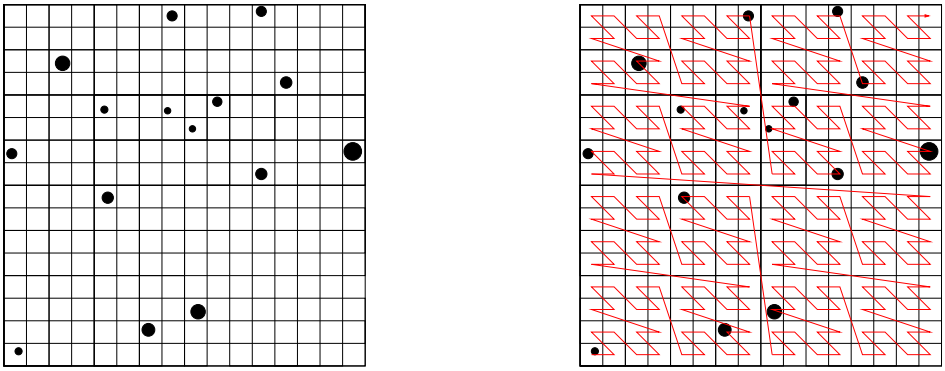


Figure 3.4: Global simulation region for a 2D particle distribution. Additionally, the local extrema are shown.

Now the space is divided into many cells and the particle position is stored in a unique binary key, by the mapping rule of a self-similar space-filling curve. The refinement of the space is shown in Figure 3.5(a).

The binary keys do not replace the coordinates but provide a natural and fast building of the tree. In PEPC a 64-bit integer is used for the storage of the keys. For any refinement level 3 bits are neccessary. Hence a total number of `nlev` $= 21$ levels can be realized with that datatype. In general, this number is sufficient to assign each particle a unique key. The mapping properties of space-filling curves are described in Chapter 4. Currently the Morton-curve is used to map the 3D values into a linear space. This curve is shown in Figure 3.5(b). An exact description of the algorithm for the generation of Morton-keys



(a) (b)

Figure 3.5: Space refinement and Morton-order for a 2D particle setup. (a) Quantized simulation region. (b) A Morton-curve covers all quanta and allocates a unique binary key for every cell.

is also depicted in Chapter 4. The range of the Morton-curve is the unit cube, so the coordinates must be normalized. The time for the calculation of a single Morton-key is independent of the underlying particle distribution. In PEPC a single key calculation consumes $6.1 \cdot 10^{-7}$ seconds on the system JUGENE.

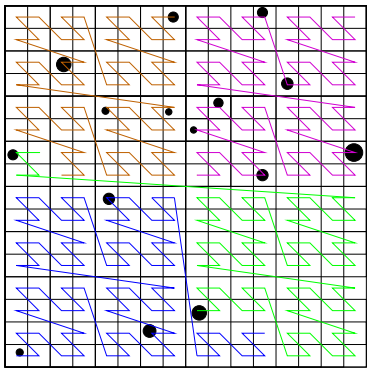


Figure 3.6: The Morton-curve is cut in pieces of varying size. These pieces are distributed among 4 parallel tasks.

Once the binary keys have been calculated and stored in `pekeys`, they will be sorted. The sorted list is distributed among the parallel tasks. As described in Section 2.2, page 10, load-balancing is an essential foundation for the performance of parallel applications. Therefore, the sort is weighted and the local particle chunks differ in size. The weights for the sort in PEPC result from the number of interactions of the particles of the previous time step. This makes sense, because the determination of the interactions, the `traversal` of the tree, represents the most time consuming part of the application.

For this reason, the local particle number differs from N/P. For simplification it was used in prior and for further considerations. Of course, sorting is done in parallel rather than on a single task.

Early versions of the code used a modification of the algorithm described in [57], which can be studied in [58]. Due to better performance, today a RADIX-sort [59] from the SL (Sorting Library) [60, 61, 62] is used instead. In this work this step will be considered as a black-box and gets no further attendance. Colloquially, the sorting is like cutting the Morton-curve in individual pieces that vary in size. This is illustrated in Figure 3.6.

3.1.2 Combination of Local Trees to a Single Data-Distributed Global Tree

In this section, the steps `local tree`, `exchange` and `global tree` of Figure 3.3 are explained. Currently, the particles are distributed among the tasks. Now a local tree can be built, according to the space division of the BH tree code. Here, the self-similarity of the space-filling curve is of crucial importance. As already mentioned, every tree node is stored in a hash table, because the number of possible keys exceeds the main memory. Furthermore, every task has a local hash table. A simple hash function is used, that generates the address for every key and maps it on the physical memory. The hash function is a simple bit-wise AND [12]

$$\texttt{address} = \text{AND}(\texttt{cellAddr}, 8^h - 1). \tag{3.1}$$

The value `cellAddr` denotes the parent-cell of the key at a specific level. This issue can be studied in Example 5.2.1. The hash function has the nice property, that for cells at a level less or equal than h, the hash address equals the key. For the other levels collision can occur. This means, two keys have the same address but a varying `cellAddr`. This count is reduced by the local address space (the local hash table) to a negligible number, since the number of local particles N/P is much lower than N. Furthermore, the access to every particle is $\mathcal{O}(1)$ instead of $\mathcal{O}(\log(N))$.

In addition to the `key` that is used for determination of the hash address, more values are stored in a hash entry:

- `owner`: This value marks the task that holds the specific cell.
- `childcode`: This value stores the state of the children cells. So it is easy to derive the child keys from the `key`. E.g. a value of $(00000100)_2$ signalises that the 2nd children cell exists (the enumeration starts with the 0th children cell).
- `link`: In the event of a collision, a `link` is set to a different hash address. By following this link, then the hash address of the collisioned key can be determined.
- `next`: A further important entry is the `next` value, which provides a pointer to the next particle simplifying the traversal of the tree.

- `leaves`: Additionally, the number of `leaves` will be stored. This value equals the number of all allocated sub cells at any lower level. A cell at any level containing exactly 1 particle is defined as leaf. A cell with 2 or more leaves is a twig.

- `node`: Another value in a hash entry is `node`. This local pointer marks twigs and leafs. Negative values mark a twig, a leaf features a positive value and 0 denotes the root cell. Through this pointer the pseudo-particle information can be accessed.

Now, the algorithm for the local tree construction is explained. All local particles will be attached to the root node at level 0. Then the space will be refined as described in Chapter 2 and in Figure 3.7. The big advantage of the HOT-scheme regarding the tree is, that the hierarchical structure is recovered automatically, because every particle key implicitly contains the neccessary information.

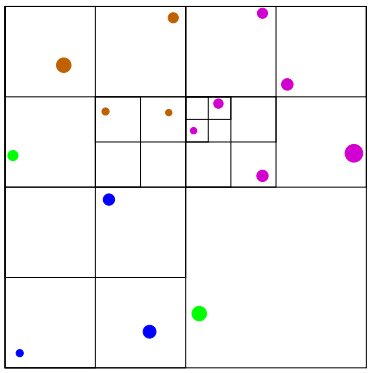


Figure 3.7: Space partioning of the parallel BH tree code. The local particles are colored regarding to their owner.

Algorithmically, the build of the local tree structure can be implemented iteratively. In Example 3.1.1 the local construction step is explained for task 0 of the previous Figure 3.7. Level by level, the appropriate key (`cellAddr`) will be evaluated for the Morton-curve of the specific level. This is accomplished with simple shift operations, enabled by the self-similarity of the space-filling curve and will be described in Example 5.2.1, page 70. From this parent-cell the hash address is calculated and a new entry is inserted into the local hash table, where several possibilities exist:

- The address is already assigned and the keys match, then a twig is detected. This twig will be stored for the next level in a todo-list, because it must be resolved at a finer resolution, to match the space division of the BH tree code.

- If a node is a leaf, that means no other keys have the same parent-cell (the same address) for the specific level, then this key gets no further attention.

- In the case of a collision, that can firstly occur for levels greater than h, a substitute address is determined and the cell is linked for further accessibility.

At a certain level, all particles are resolved into leaves. As a reminder, in cells that contain only one particle. This is the termination condition, which can be recognised by an empty todo-list. Empty cells are intrinsically eliminated, because these are not contained in any particle key. By the gradual construction, additional cells, so called fill nodes are introduced. For a homogeneous setup, their number can be approximated by $1/7 \cdot N/P$. That is the limit of $\lim_{l\to\infty} N/P \sum_{i=1}^{l} 1/8^i$. We suppose that all particles are on the level l. Then for the next lower level the number of fill nodes is 1/8 and so on. For the inhomogeneous case the number can be estimated by N/P. An example in [18] demonstrates the worst case, where any single particle introduces an extra fill node. All cells are stored in the array `treekeys`. The hierarchical view of the local data structure, which is insufficient for the essential `force summation`, is illustrated in Figure 3.8. In a nutshell, every task has its own local tree and hash table and the properties of some local nodes may be incomplete.

Example 3.1.1. The local tree construction is shown for task 0. The maximal refinement level is `nlev` = 4. The task owns 3 local particles with the keys: $(10000)_4$, $(10112)_4$ and $(10323)_4$ (an additional 1 for level 0 was added to describe the root node, the last 4 bits can be derived by Figure 3.7). For the 1st level the parent cell (`cellAddr`) for all keys is: $(10)_4$. Since this cell contains 3 particles it must be refined and the keys are put on the todo-list. Here, a fill node was introduced and inserted in the hash table with the properties: `key` = $(10)_4$, `owner` = 0 (task 0) and `leaves` = 3. Since the level is lower than h the key equals the hash address. For the 2nd level the parent cells (`cellAddr`) are: $(100)_2$, $(101)_2$ and $(103)_2$. All parent cells differ, therefore each cell only contains 1 particle, and the refinement is finished. Thus the todo-list for the next level is empty, which indicates that the local construction is finished. All cells are inserted in the hashtable with the properties: `key` = $(\cdot)_4$, `owner` = 0 (task 0) and `leaves` = 1. Furthermore, the chilcode of the fill node $(10)_4$ of the previous level is updated: `childcode` = $(1011)_2$.

Once the local part is formed on each task, all trees have to be combined into a single global structure. For this global tree the data is distributed, since a fetch of all non-local nodes would ruin all benefits of a parallel tree code, regarding memory consumption (less particles can be simulated) and scalability. Without fetching all non-local nodes, the local trees of all other tasks are integrated in another way into the local tree (local hash table). Every local tree can be described by a few nodes, that overlap the entire structure. These nodes will be interchanged between all tasks (step `exchange` of Figure 3.3) and integrated in every local hash table. This nodes are called branch nodes or branches. Below the branches every task only contains the local nodes. If non-local nodes are required, the appropriate branch provides the `owner` and the task can request this information. Above the branches, the local trees can contain incomplete twigs, which are also shown in Figure 3.8. That are twigs crossing domain boundaries, as shown for task 1 and 2. Also the root node of every local tree needs information from every other task.

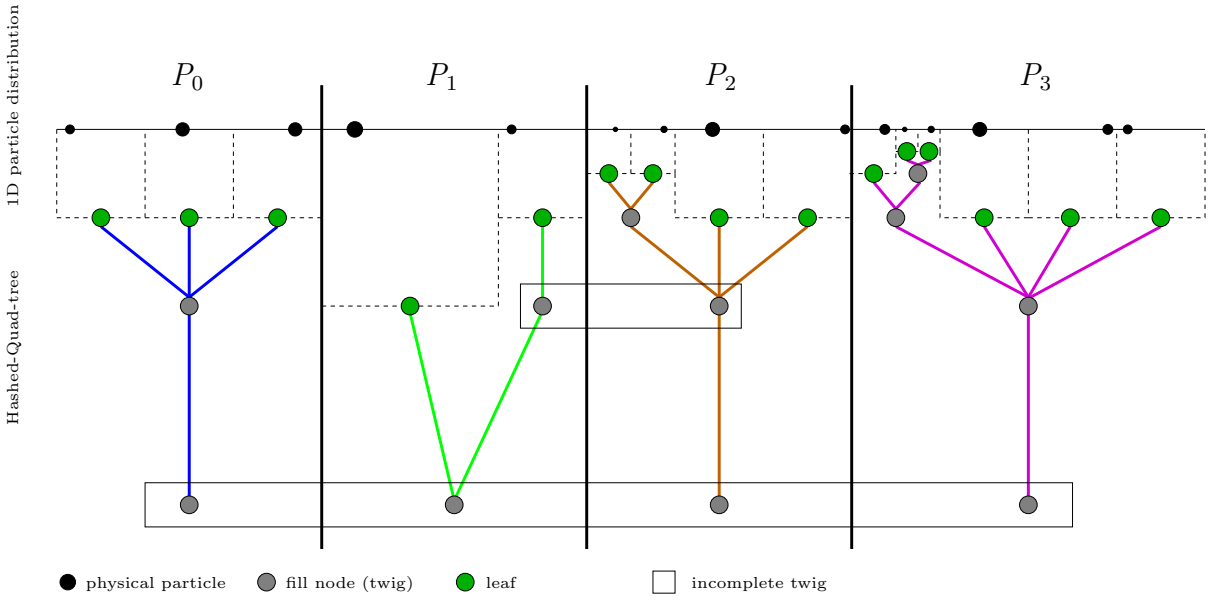


Figure 3.8: Hierarchical view of the data structure after the local trees were built. Each task has its own local tree that is labeled with the appropriate color. In the current status incomplete twigs exist. E.g. the root node is shared by all tasks and includes only partial data for every task.

The set of branches is the smallest set of keys, that covers the entire local domain, and only the local domain. A single branch node possesses the following characteristics.

Definition 3.1.2. A branch is an administrative unit, that equals a node in the tree (here 3D). It will be used to assemble a global tree out of P local trees and as a gate to non-local information in a data-distributed global tree. A branch has following characteristics:

(i) It is on a fixed level: $0 \leq \texttt{level} \leq \texttt{nlev}$.

(ii) It includes $8^{(\texttt{nlev}-\texttt{level})}$ cells of the level `nlev`.

(iii) It covers an area in the linear key space:

$$c \cdot 8^{(\texttt{nlev}-\texttt{level})}, \ldots, (c+1) \cdot 8^{(\texttt{nlev}-\texttt{level})} - 1, \;\; c \in \left\{0, \ldots, 8^{\texttt{level}} - 1\right\}.$$

and every node below the branch must have the same `owner`.

The set of branches for the specific example is geometrically illustrated in Figure 3.9. The hierarchical view on the state of the data structure, that includes the branch nodes, can be seen in Figure 3.10.

The current branch concept is far from being ideal, which can be seen Figure 3.10, because almost every leaf node has been detected as a branch. E.g. in Figure 3.9, both small brown branch nodes could be combined, but the original branch concept provides

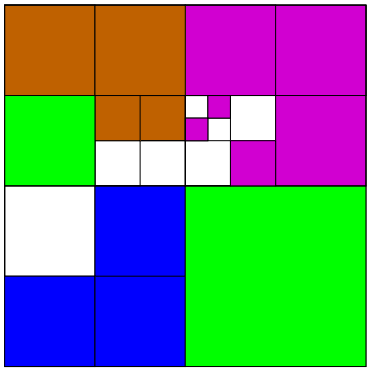


Figure 3.9: Geometrical view on the branch nodes for 4 tasks in a 2D example.

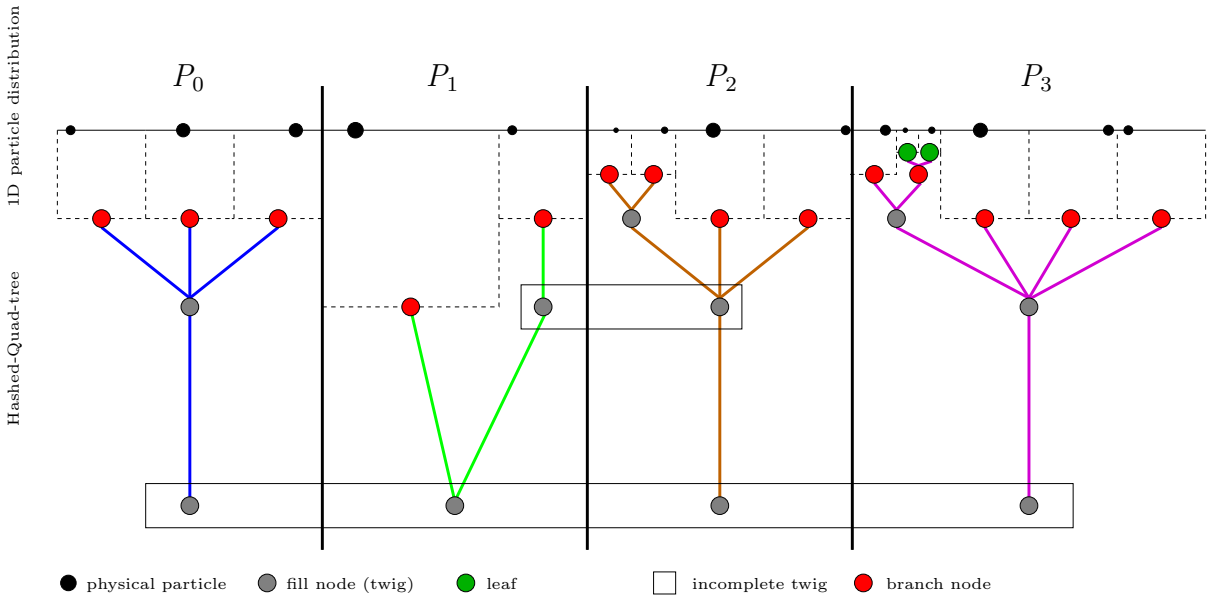
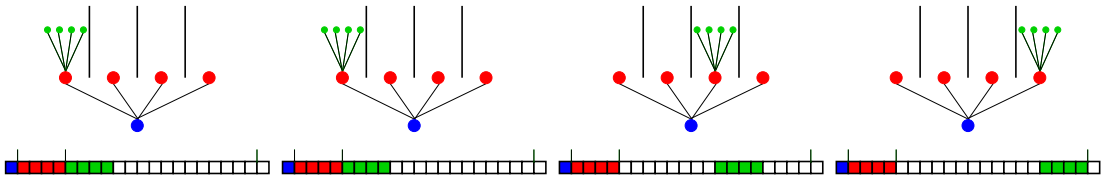


Figure 3.10: Hierarchical view of all local trees, after the branch nodes are known. A large amount of branches regarding to the number of leafs was detected with the original finding algorithm.

both nodes. To circumvent this small branch nodes and to minimise the amount, a novel concept will be introduced in Chapter 5. In Example 3.1.3 the local branch nodes are determined for task 0.

Example 3.1.3. The local tree for task 0 consists of the incomplete root node: $(1)_4$, the fill node $(10)_4$ and the leaves $(100)_2$, $(101)_2$ and $(103)_2$. The original branch concept brings 3 branch nodes, the leaves of the local tree. This nodes cover the entire local domain and not any particle of another task. Since the cell $(102)_4$ is empty and does not belong to any other task, already the fill node $(10)_4$ fulfills every branch condition (see Definition 3.1.2). In Chapter 5, a novel concept is introduced, which determines exactly this fill node as the only branch for the local domain of task 0.

The current status of the data structure is: the local trees were built, the local branches were determined and integrated in every local hash table (local tree), but above the branches incomplete twigs could be exist. Firstly, the local fill nodes above the branches are disregarded. Secondly, these nodes are built afresh from branch level up to root for every task, and stored in each local hash table. This automatically merges the incomplete twigs to complete nodes in a global context. Now every task features the same global tree up to the branch level. For deeper level only the local information is available. In Example 3.1.4 the fill process is depicted for the incomplete twig between task 1 and 2 and an abstract view of the global data-distributed structure is shown in Figure 3.11.



(a) task 0 (b) task 1 (c) task 2 (d) task 3

Figure 3.11: Data-distributed global tree and the local part, with the respective hash entries, for 4 tasks. The root node is marked with a blue circle, global branches are red colored. The local particles are green.

Example 3.1.4. The keys of the nodes can be achieved by the evaluation of Figure 3.6 or Figure 3.9. One node (leaf) of task 1 (green), namely the key $(120)_4$ and all nodes of task 2, namely $(1212)_4, \ldots, (123)_4$ (brown), share a common fill node $(12)_4$. In every local hash table only the local nodes influence the values of the local hash entry. For task 0 this fill node has 1 leaf and `childcode` $= (0001)_2$. For task 2: `leaves` $= 5$ and `childcode` $= (1110)_2$. In a global context, this fill node only exists once. For the combination of the incomplete local fill node, which is present in both local hash tables of task 1 and 2, the `leaves` are added and the `childcode` is linked by an $\textsc{Or}(\cdot,\cdot)$ operation.

Hence, for the global fill node, that is shared by task 1 and 2, the following properties result: `leaves` $= 1 + 5 = 6$ and `childcode` $= \textsc{Or}((0001)_2, (1110)_2) = (1111)_2$.

The hierarchical view is illustrated in Figure 3.12 and the shared part of the data-distributed global tree is marked with black lines. For task 3 the local part can be seen. For the other tasks, depending on the small particle number in this example and the original branch concept, all leaves are resolved in branches. Furthermore, a comparison with a serial HOT-scheme is shown hierarchically. In this case, the concept of branch nodes is not needed and incomplete twigs do not exist.

Once the global data structure is in place, it is a straightforward matter to calculate the pseudo-particle properties for each node, that will be accessed by the hash entry `node`. For every twig the calculation of the properties is simple, because the multipole moments can be successively shifted up to their parent level using displacement vectors [4].

3.1.3 Tree Traversal and Force Summation

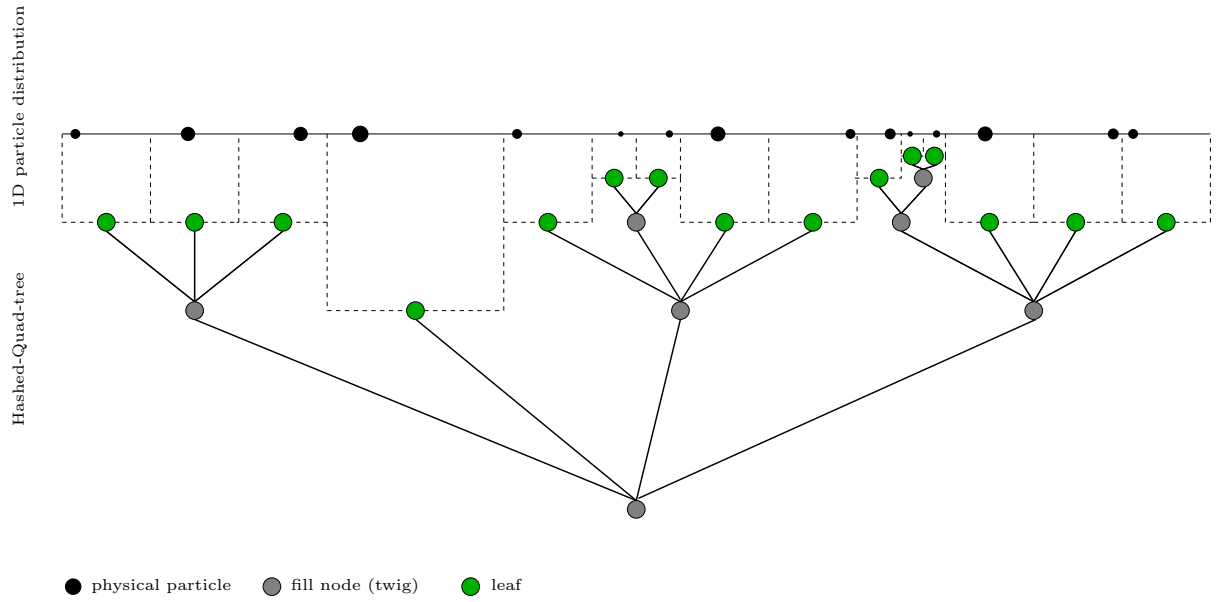
At the `traversal` and the `force summation` stage, a global virtual tree (see Figure 3.11)is known on each task, and the pseudo-particle information is attached to each tree node. The purpose of this step is the evaluation of the pair potential, which is used in the front-end for pushing the particles to another position.

Now the global tree will be traversed for every local particle. At every tree node, starting with root, the acceptance criterion is checked and two possibilities exist:

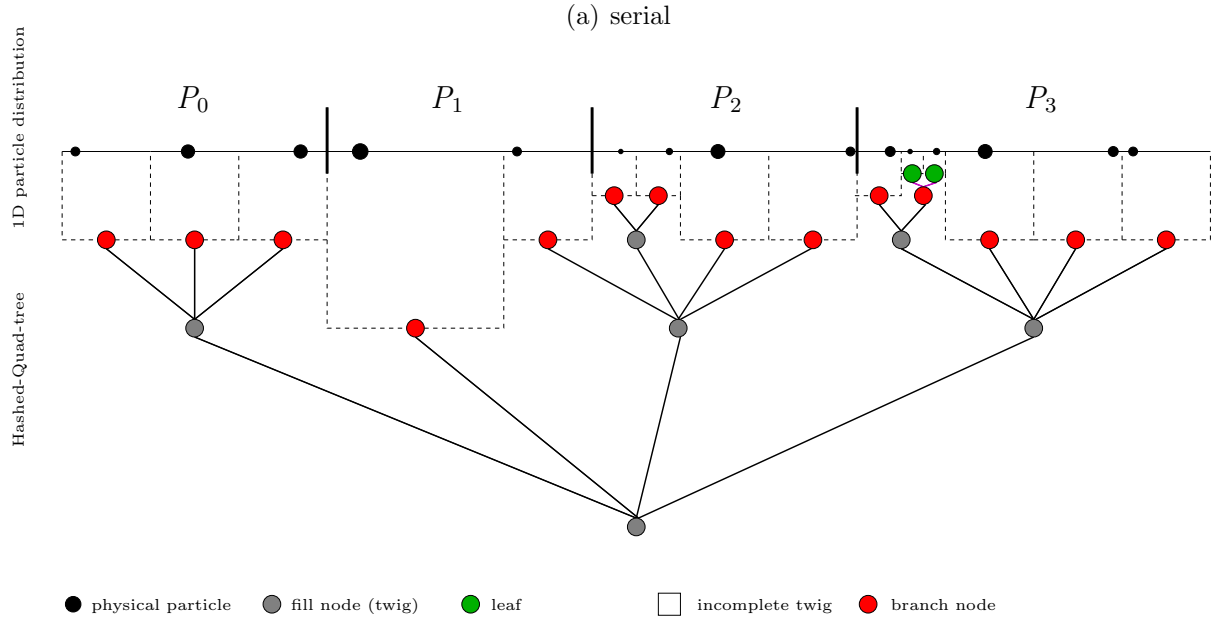
- If the criterion is fulfilled, then the particle interacts with this node. Now the particle continues its traversal with the `next` key, which is stored in every hash entry.
- Otherwise, the node is resolved into its children, and the criterion is checked for exactly this nodes. The determination of the child nodes is simple, because the hash entry saves the appropriate information (`childcode`). With the child key, the hash address is calculated, and the information can be accessed with $\mathcal{O}(1)$.

After all particles have finished their traversals, all interactions, particle-particle (leaf node), or particle-pseudo-particle (twig node) are available. This is a serial HOT-scheme. The problem is, that each task only has a virtual global tree. As a reminder, down to the branch level any information is globally known. Beyond the branch level, each task only knows its own local tree. Therefore, in a parallel HOT-scheme attention must be paid to the availibilty of the data:

- The traversal reaches a node, where no pseudo-particle informations is locally known. Firstly, this happens for the children of branch nodes. For the purpose of the evaluation of the acceptance criterion, the information must be requested from the `owner`, which is also stored in a hash entry. The hash table of the owner includes



(a) serial



(b) parallel

Figure 3.12: Comparison of the hierarchical view of the global tree for the serial and parallel HOT-scheme. The global fill nodes have been added. In (a), the concept of branches is not needed, because the global structure is in place automatically. (b) For the parallel version, the black lines show the part of the global tree that is shared by all tasks. The red colored branch nodes act as a gate to non-local information (local tree of another task, below the branch level, and thus to its local hash table). In the example this only can be seen for task 3, where the local part below the branch is combined with violet lines. The incomplete twigs are not present anymore.

the pseudo-particle information. After the owner has send the information, the acceptance criterion can be checked and it is continued as described before. Since the possibility is high, that other local particles need this non-local information, it is stored in the local hash table.

- Either the current node belongs to the local tree or the node information has already been requested for another particle and added to the local hash table, the criterion can be evaluated.

On the one hand, a difficult memory management is the consequence. This is because the requested nodes are stored locally and their number cannot be accurately predicted. On the other hand, an irregular communication structure emerges and only point-to-point communication is suited instead of collectives. For this reason, by far the most challenging part in the parallel HOT-scheme is the `traversal`. In this thesis two concepts, which are implemented in PEPC, are introduced [63]:

- Pure MPI: In practice, chunks of particles perform a simultaneous traversal. If any information is missing, then the non-local node is placed on a list, and the traversal proceeds with another particle. After all particles have made their check of the acceptance criterion for exactly one node, the nodes in the list are worked off and the required information is requested from the `owner`. When the communication has finished, the requested node information is locally available and the criterion can be evaluated. Every interaction is stored on an interaction list, which will be summed up in a separate `force summation`. In contrast to the direct algorithm, where the interaction count is exactly $N(N-1)/2$ for every particle, here the interaction list length must be estimated. This causes a large memory footprint.

- MPI-Pthreads: In this hybrid approach one communicator-thread per MPI task handles the request of non-local information and several worker-threads perform the `traversal` for each particle. If any non-local node information is required, the communicator-thread is started. This thread determines the `owner` and handles the communication. After the information has arrived, it notifies the worker-thread, which has requested the information.
  Any worker-thread traverses the tree for an individual set of particles. If non-local information is required, it transmits the request to the communicator-thread and processes with another particle. In this approach, the force is directly calculated if an interaction was determined. Hence, the interaction list for each particle can be avoided.

The hybrid ansatz with several worker- and a single communicator-thread has a lot of advantages. Firstly, the interaction list can be avoided, and in this context less memory must be allocated, because the force of the individual interaction will be directly calculated. Furthermore, the throughput will be increased, because a single thread handles

the information requests and the worker-threads are simultaneously calculating the force. This is a nice overlap of communication and computation. The previous explanations are only a snapshot of this very complex approach, but for the main topic of this thesis, a deeper insight is not neccessary.

3.2 Scaling and Determination of Bottlenecks

For benchmarking, the hybrid tree traversal approach was used, because this ansatz is up to date and many measurements exists. Here the number of cores is $4 \cdot P$, because 4 threads were used in SMP mode on JUGENE. In this case the maximal available number of MPI tasks is 73728. A differentiation between threads and tasks was made in Section 2.2, page 10.

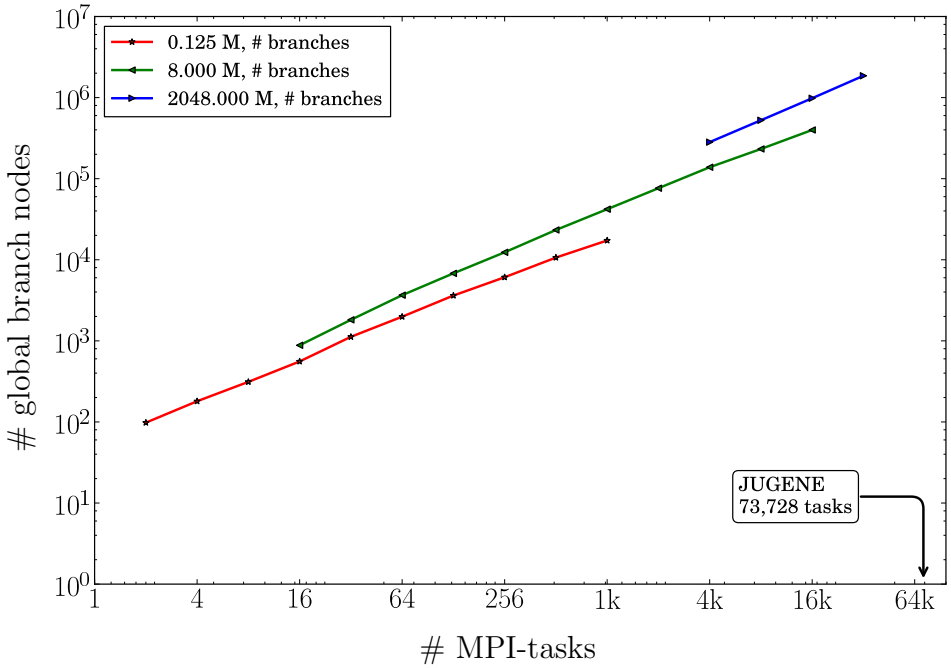


Figure 3.13: The number of global branch nodes for the inhomogeneous setup. Since no significant deviation to the homogeneous case exists, only this setup is shown. Furthermore, the number of global branches depends on the number of local particles.

The main bottleneck arising from a data-distributed tree code is the combination of P local trees to a virtual global structure. For this reason, the concept of branch nodes was introduced. It is a straightforward matter to keep the number of these nodes on a minimum level. For this problem, the number of MPI tasks P is essential and scalability will profit by a decreased number of branch nodes. The `exchange` is made by an

MPI_ALLGATHERV. This collective routine is applied for every task and gathers data of individual size from each other task. Naturally, this routine profits from a decreased number of branches alike, but gets worse for an increasing number of tasks at large scales. Being far from ideal, in Figure 3.13 the weakness of the old branch concept is underlined for the inhomogeneous setup. Here, the global number is shown for different tasks, that depends on P naturally, since the local domain must be covered and at least one branch node is needed per task. Furthermore, the conclusion can be derived, that the global number depends on the average number of local particles N/P. This is because the curves for higher local particle numbers are positioned above curves with a lower one. This effect intensifies the bottleneck extraordinarily. As a result, for $P \gtrapprox 32000$ and $N = 2048 \cdot 10^6$ the global branch amount blasts the available memory. Inferentially, the original branch concept prevents parallel N-body simulations with PEPC at extreme scales, because all global branch nodes must be stored locally. In addition, no significant differences between the homogeneous and inhomogeneous setup can be identified. On this account the number of global branch nodes is only shown for the inhomogeneous setup.

In a serial version of the HOT-scheme the branches do not exist in the same way, see Figure 3.12(a). They are already included as fill nodes, but do not have to be determined and exchanged. So one can confidently describe their construction and exchange as pure parallel overhead. The immense effect of the branch exchange on the overall run time can be seen in Figure 3.14.

At small scales the amount of branches is low [64] and the `exchange` contributes an unnoticed percentage (for $P = 256$ and $N = 8 \cdot 10^6$: $\approx 1\%$) to the total runtime, which is dominated by the `tree traversal` and the `force summation`. Since these steps scale well, and in the same account the `exchange` increases it becomes a major part of the run time. Finally, the `exchange` takes longer than the `tree traversal` and the `force summation` (e.g. for $P = 16384$ and $N = 8 \cdot 10^6$: $\approx 60\%$ of the total run time). Hence, for any particle number at a cross over point, the total run time increases and speedup is prevented.

Another bottleneck regarding to the memory consumption, is the size of the local hash table, which cannot be predicted. In the current implementation it is roughly estimated. Hence costly memory is wasted and the storage for possible particles remains empty. If a tight estimation of the memory amount of the hash table would be available, then this issue could be optimised. Concerning the hash table entries, they are of the following kind:

- Leaf nodes are exactly known by N/P.
- Fill nodes, which can be estimated, as described in Section 3.1.2 by $1/7 \cdot N/P$ (homogeneous) and N/P (inhomogeneous).
- Global branch nodes. Their number cannot be tightly forecast and highly increases for large scales.

- Nodes which were requested during the `tree traversal`. This number cannot be estimated in a satisfactory way.

One reason for a deficient memory estimation are the global branch nodes. In a nutshell, the branch nodes limit the increase of the particle number, the scalability of the parallel BH tree code PEPC and the execution at extreme scales.

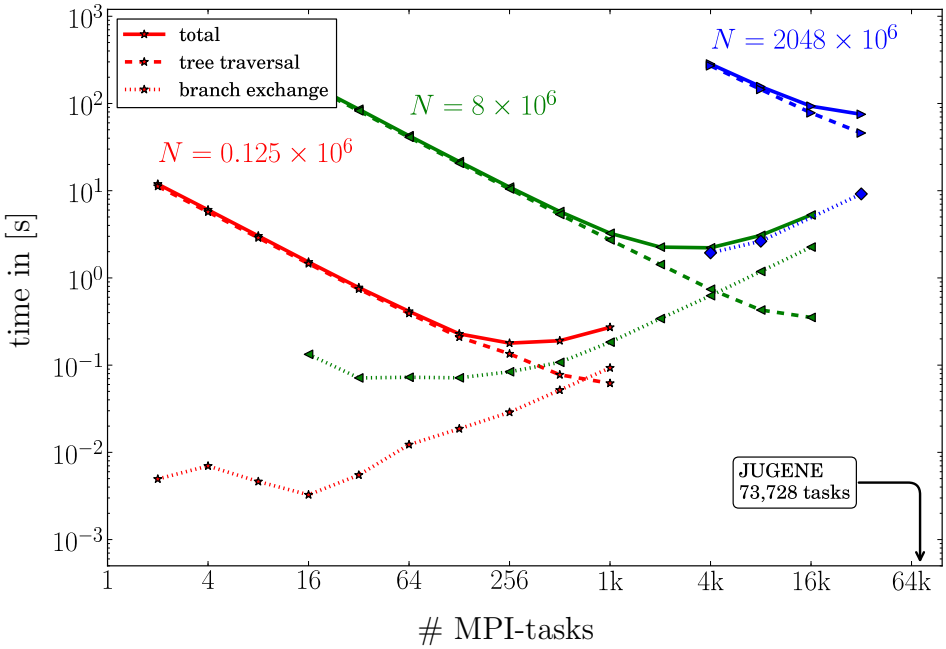


Figure 3.14: Scaling of the code for the inhomogeneous setup. Here, the major algorithmic bottleneck of the HOT-scheme, namely the `exchange`, can be seen directly. The dashed lines represent the time of the combined `traversal` and `force summation` in a hybrid version of the code. The dotted lines show the time for the branch node `exchange`. These steps are the dominating contributions to the total runtime (solid lines). No significant differences can be found between the homogeneous setup the inhomogeneous one. Hence only the scaling for the inhomogeneous setup is shown.

Another problem naturally emerging is the determination of the branch nodes, which is currently implemented very poorly. Besides the original branch concept, the algorithm produces much unnecessary overhead. However it contributes only a very small fraction to the total run time, since each task only must find its local branches. The scalability and memory footprint of the other steps in the HOT-scheme are descibed below:

- The key construction in the `domain decomposition` scales linearly, because the number of local particles decreases for enlarged task numbers.

- With the same argument, the `local tree` construction scales. With a sophisticated implementation no further memory must be allocated.
- The step `global tree`, where the global fill nodes (the global nodes which are shared by each task) are calculated and the pseudo-particle information is attached to each node, scales. Indeed, it depends on the deepest branch level and the original branch concept highly depends on the local particle number and the number of tasks, so the branch level increases. But the contribution of this step to the total run time is very small, and a further treatment does not yield any measurable improvement. The memory footprint vanishes in either case.
- For an appropriate θ and effectual high local particle number the `tree walk` and the `force summation` scales. Perhaps the run time of this routine, which is a major part of the total run time, can be reduced by the application of alternative space-filling curves. Even the hybrid ansatz needs less memory allocation, because the interaction list is avoided.
- Another step is the weighted parallel sort in the `domain decomposition`. This step depends on the number of particles and tasks, and on the distribution of the particles. The disregard of the sorting routine in this thesis can be motivated by the light contribution of this step to the total run time, which is magnitudes smaller than the branch exchange.

The following chapters intensively deal with the minimization of the number of branch nodes as a integral bottleneck, to overcome a major obstacle of a data distributed parallel BH tree code.

Chapter 4

Application of the Hilbert-curve to Tree Codes

As seen in the previous chapter, the HOT-scheme needs a fast mapping from the 3D coordinate values to universal binary keys in 1D (`domain decomposition` in Section 3.1.1, page 23). Curves with a space-filling behaviour, so called SFC (Space-Filling Curves), are highly qualified to deliver such a fast and unique mapping.

The brief history of SCF is outlined in the following. Arround the turn of the 19th century, before any computer was built, the existence of curves that pass through all points of higher-dimensional space was found. In 1890 Giuseppe Peano constructed the first SFC [65], David Hilbert in 1891 [19], Eliakim Hastings Moore in 1900 [66] and Henri Leon Lebesque in 1904 [67] are only a subset of famous successors. The space-filling behaviour had fascinated mathematicians for over a century. Today they are an elemental ingredient in many codes. Among topics like image processing, e.g as a scan line for path finding [68] or the storage and retrieval of multi dimensional data [69], tree codes like the FMM and the BH tree code are splendid examples of application of SFC in modern times.

For the general introduction of SFC that follows the one advised by [20], firstly two requirements of measure theory have to be introduced.

Definition 4.0.1. If $f : (\mathbb{R}^n, \|\cdot\|_2) \supset A \to f(A) \subset (\mathbb{R}^m, \|\cdot\|_2)$ then the direct image of f is defined as follows:

$$f_*(A) = \{f(x) \in \mathcal{R}(f) : x \in A \cap \mathcal{D}(f)\}.$$

$\mathcal{D}(f)$ denotes the domain, and $\mathcal{R}(f)$ the range of f. If $f : [0,1] \to (\mathbb{R}^m, \|\cdot\|_2)$ is continuous ($n = 1$ and $A = [0,1]$), then the direct image of f, $f_*([0,1])$, is called a curve.

For the measurement of a bounded subset of $\mathbb{R}^m$ the Jordan content can be used. The subset is filled up with disjoint rectangles, $R \in \{]a, b[: a, b \in \mathbb{R}^m, a \leq b\}$, whose union is called a simple set: $\bigcup_{k=1}^{l} R_k$. The subset $A \subset \mathbb{R}^m$ can be approximated both, from the inside, which is the inner Jordan measure $J_*(A)) = \sup_{S \subset A}(J(S))$, and the outside, which is the outside Jordan measure $J^*(A) = \inf_{S \supset A}(J(S))$. If the inner measure equals

the outer measure, the subset is Jordan measurable and the value is called the Jordan measure $J(A)$ [70, 71]. That is like the Riemann integral for higher dimensional sets. Because $J(A)$ is not σ-additive it is only a content and no measure in the usual sense. With these requisites, SFC can be generally defined.

Definition 4.0.2. If $f : [0,1] \ni x \to f(x) \in (\mathbb{R}^m, \|\cdot\|_2)$, $m \geq 2$, is continuous and has a positive Jordan content $J_m(f_*([0,1])) > 0$, then the direct image of $[0,1]$ under f, $f_*([0,1])$, is called a space-filling curve.

In tree codes the inverse map from 3D to 1D, thus f^{-1} and the fast algorithmic computation of so called derived keys is of interest. Eugen Netto proved that bijective maps between smoothed manifolds are neccessarily discontinous [20]. An SFC is a continous map, thus it can not be bijective. Since every point is passed, it is leastwise surjective and not one-to-one.

In practice, the mapping must be invertible and for a finite number of the derived keys this should be unique. Therefore, the curve must not neccessarily pass through every point in space, but through a finite number of points. This leads to a quantized space and approximations to SFC $\tilde{f}$. The type and the degree of quantization depend on the application. In the 3D BH tree code, every direction is halved for each refinement step. So the number of points, at a fixed refinement level, is $2^{3\texttt{level}}$. The domain of $\mathcal{D}(\tilde{f})$ then becomes:

$$\mathcal{D}(\tilde{f}) = \left\{0, \frac{1}{2^{3\texttt{level}}}, \ldots, \frac{2^{3\texttt{level}}-2}{2^{3\texttt{level}}}, \frac{2^{3\texttt{level}}-1}{2^{3\texttt{level}}}\right\} \tag{4.1}$$

Further, the unit cube is taken as the image space. From the discrete set of points of $\mathcal{D}(\tilde{f})$ a discrete realisation of a space-filling curve $\mathcal{S}^3_{\texttt{level}}$ to the discrete image $\mathcal{R}(\tilde{f})$ can be found.

$$\mathcal{R}(\tilde{f}) = \left\{\left(\frac{i-1}{2^{\texttt{level}}}, \frac{j-1}{2^{\texttt{level}}}, \frac{k-1}{2^{\texttt{level}}}\right)^{\mathsf{T}}, \quad i,j,k = 1, \ldots, 2^{\texttt{level}}\right\} \tag{4.2}$$

Obviously, for $\texttt{level} \to \infty$, $\mathcal{D}(\tilde{f}) \to [0,1]$ and $\mathcal{R}(\tilde{f}) \to [0,1]^3$. Therefore, the discrete realisation $\mathcal{S}^3_{\texttt{level}}$ built on the quantized space converges towards a space-filling curve $\mathcal{S}^3_\infty \equiv \mathcal{S}^3$ for $\texttt{level} \to \infty$ (means $\lim_{i\to\infty} \mathcal{S}^3_i = \mathcal{S}^3$). For completeness, other curves exist that have a triadic quantization of the space, e.g. the Peano-curve [20, 65]. The Sierpinsky-curve passes through triangles [20, 72].

Notation 4.0.3. For $m \geq 2$, $\texttt{level} \geq 1$ let $\mathcal{S}^m_{\texttt{level}}$ be the m-dimensional-$\texttt{level}$-generation, the discrete realisation of $\texttt{level}$ or $\texttt{level}$-iteration of a space-filling curve, that is denoted with $\mathcal{S}^m_{\texttt{level}}$. This curve maps from $[0, 2^{m\texttt{level}}]$ into $[0, 2^m]^{\texttt{level}}$.

To increase the simplicity only the numerator of the points in $D(\tilde{f})$ is used as derived key. Likewise the numerator of $R(\tilde{f})$, the so called partial keys (i_z, i_y, i_x in 3D), is taken

for the computation of the derived keys via $\tilde{f}^{-1}$. For general N-body systems, that are not merely arranged in the unit cube, the partial keys can be normalized to the unit cube.

For this thesis, the whole set of SFC is reduced to those that are FASS-curves (space-filling, self-avoiding, simple and self-similar):

- Particularly the self-similarity is a neccessary requirement for an efficient HOT-scheme, because the required information for the tree can be obatined from the key. Self-similarity means that the portion of the curve on the interval $[k - 1/2^{m\cdot\texttt{level}}, k/2^{m\cdot\texttt{level}}], \texttt{level} = 1, 2, \ldots, k = 1, 2, \ldots, 2^{m\cdot\texttt{level}}$ is an exact replica of the entire curve, that is reduced in the ratio $2^{\texttt{level}} : 1$ [20]. The passage from $\mathcal{S}^m_{\texttt{level}}$ to $\mathcal{S}^m_{\texttt{level}+1}$ is researched. Every quantum of the quantized space is again divided in 2^m quanta (in 3D every octant is divided in 8 new octants). In every quantum a curve like $\mathcal{S}^m_{\texttt{level}}$ is put. These 2^m curves are combined to $\mathcal{S}^m_{\texttt{level}+1}$ (in 3D eight curves $\mathcal{S}^3_{\texttt{level}}$ are combined to $S^3_{\texttt{level}+1}$).

- Self-avoidance means that a point which is passed by the curve is never reached again. This issue induces injectivity (for a quantized space).

- The attribute space-filling depicts the surjectivity of a FASS-curve.

Here premarily byte-orientated algorithms are considered. To represent octants in 3D, that are numbered with $0, \ldots, 7$, exactly 3 bits are neccessary $((000)_2, \ldots, (111)_2)$. In m dimensions m bits are needed. In 3D the tripels are combined to a binary digit, where higher order triples describe the octants at a less refined space. This is depicted in Example 4.0.4. For general quantized spaces it works analogously, but with a union of bit-sets that include m bits. Self-similarity guarantuees that the bit triple, the bit sets of size m, respectively, are independently of one another. Assuming a fixed maximal refinement level `nlev` then $3 \cdot \texttt{nlev}$, or $m \cdot \texttt{nlev}$ bits are neccessary in 3D or mD. The partial keys are composed out of `nlev` bits each. An infinite-sized binary digit can describe the SFC for a non-quantized space, that means the "real" SFC and no approximation, but is not available and not needed in practice. If one takes an approximation higher bits than $3 \cdot \texttt{nlev} - 1$, or $m \cdot \texttt{nlev} - 1$ are zero.

Example 4.0.4. The aim of this example is the explanation of the independence of the refinement levels for a FASS-curve. A randomly chosen key of any FASS-curve (3D) is shown.

$$(\underbrace{110}_{\texttt{l=1}}\underbrace{001}_{\texttt{l=2}}\underbrace{111}_{\texttt{l=3}}\underbrace{101}_{\texttt{l=4}})_2.$$

The first triple $(110)_2$ means that the key is lying in the 6th octant of a 1st order curve $\mathcal{S}^3_1$. The next triple $(001)_2$ describes, that the key lies in the 1st octant of the 6th 1st order octant. That is the $6 \cdot 8 + 1 = 43$th 64-tant of the 2nd-order curve $\mathcal{S}^3_2$. The 3rd triple $(111)_2$ depicts the 7th octant of the 1st 2nd order octant thereof.

In the following, 2D examples are considered to improve the geometric clearness. Two curves are fulfilling the contrived requirements. The Morton-curve, on the one hand, is currently implemented in PEPC. The Hilbert-curve, on the other hand, is integrated in line with this thesis. It needs an efficient implementation in the parallel tree code, that is described in the adjacent section. This curve features a better data locality, that means a better spatial locality in the linear space. Therefore, it promises a benefit for the tree code at various stages. In the paper [22] it was analytically and empirically shown that the Hilbert-curve achieves the best clustering. For both curves, the geometrical generation and fast byte-orientated algorithms are described in the following. The focus is on the generation of derived keys in 1D, means the mapping by $\tilde{f}^{-1}$.

4.1 The Morton-curve

The Morton-curve was introduced in 1966 by G. M. Morton [21]. It is also known as Z-curve but not to be confused with a method for genome analysis [73]. The name comes from the fact that many z-shaped patterns are combined to a curve. In Figure 4.1, the first 4 iterations $\mathcal{Z}_1^2, \ldots, \mathcal{Z}_4^2$ of the Morton-curve are shown.

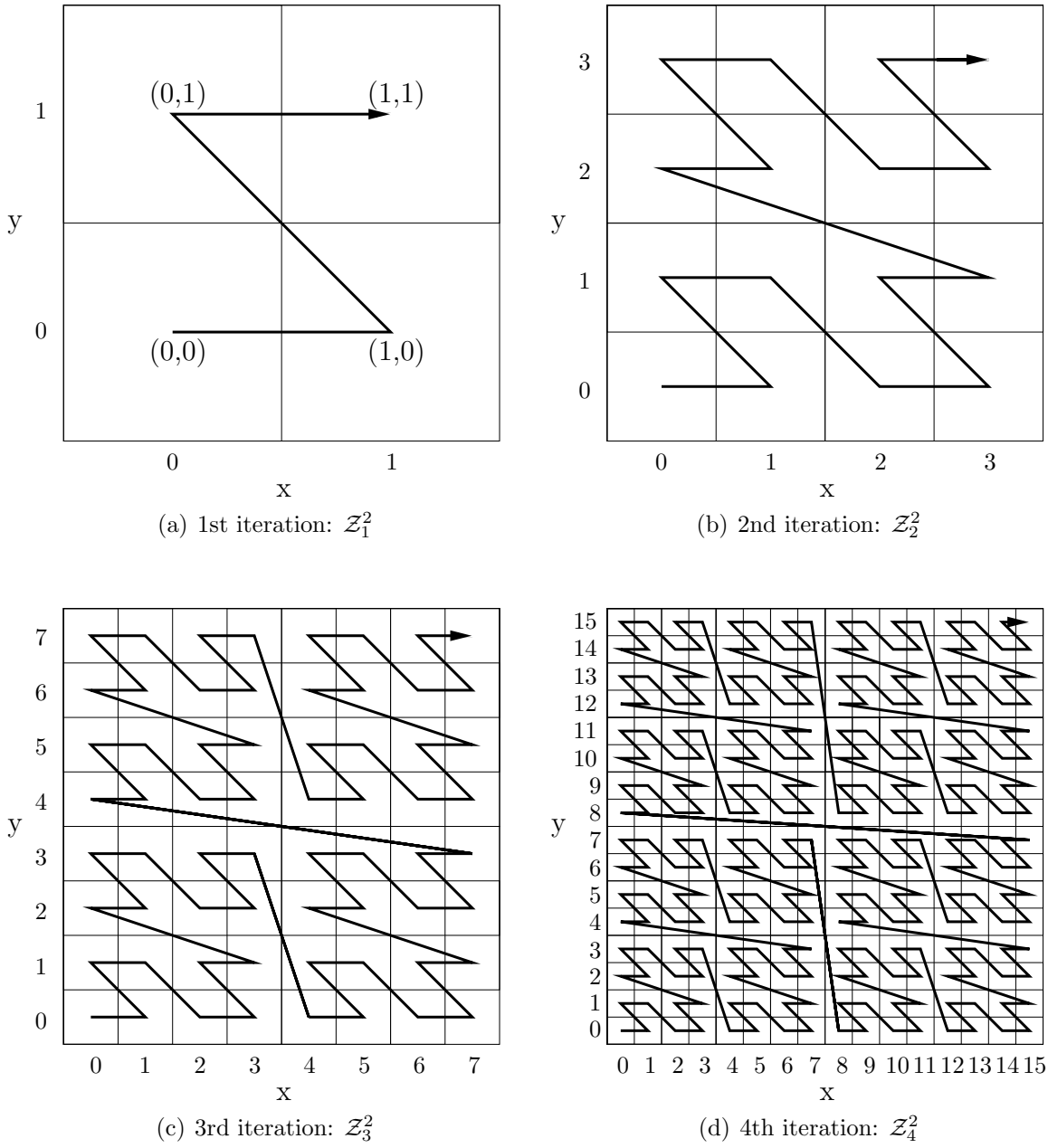
It is a FASS-curve which has the pretty advantage that exactly the same curves $\mathcal{Z}^m_{\texttt{level}}$ are combined to the higher order curve $\mathcal{Z}^m_{\texttt{level+1}}$ at the next refinement level. In 3D in every octant of the next refinement level the curve $\mathcal{Z}^3_{\texttt{level}}$ is put and combined to $\mathcal{Z}^3_{\texttt{level+1}}$. Hence the Morton-derived key provides a simple and fast mapping – the binary interleave operation of the partial keys.

4.1.1 Inverse Mapping from mD to 1D

Algorithmically, the binary interleave operation can be straighforward implemented. The partial keys $i_{x_1}, \ldots, i_{x_m}$ are given. Every of these coordinates can be described with `nlev`-bits. Then, for each refinement level the highest bit of the partial keys is truncated and combined to the appropriate quantum. The sequence of the combination is $(i_{x_{m,\texttt{nlev-level}}}, \ldots, i_{x_{1,\texttt{nlev-level}}})_2$, where $i_{x_{a,b}}$. denotes the b-th bit of the a-th coordinate. This step is performed for all refinement levels, and the quantum for the level 1 is placed at the highest position of the key. In Equation 4.3 the key for the 3D case is listed.

$$(z_{\texttt{nlev}-1} y_{\texttt{nlev}-1} x_{\texttt{nlev}-1} z_{\texttt{nlev}-2} y_{\texttt{nlev}-2} x_{\texttt{nlev}-2} \cdots z_1 y_1 x_1 z_0 y_0 x_0)_2 \tag{4.3}$$

Since one loop over all bits of the partial keys is needed the complexity is of the order $\mathcal{O}(\texttt{nlev})$. An FORTRAN90 implementation can be found in Appendix A.1 and a 3D example is composed in 4.1.1.



(a) 1st iteration: $\mathcal{Z}_1^2$

(b) 2nd iteration: $\mathcal{Z}_2^2$

(c) 3rd iteration: $\mathcal{Z}_3^2$

(d) 4th iteration: $\mathcal{Z}_4^2$

Figure 4.1: Geometrical construction of the $\mathcal{Z}$-curve in 2D. The first 4 iterations $\mathcal{Z}_1^2, \ldots, \mathcal{Z}_4^2$ are shown. According to the level of the curve $1, \ldots, 4$ the partial keys $i_y = i_x = 0, \ldots, 2^{\texttt{level}} - 1$ are marked.

Example 4.1.1. We take a 3-dimensional-3rd-generation Morton-curve $\mathcal{Z}_3^3$ (`nlev=3`) and a arbitrary chosen coordinate triple: $i_x = (x_2x_1x_0)_2 = (000)_2$, $i_y = (y_2y_1y_0)_2 = (111)_2$ and $i_z = (z_2z_1z_0)_2 = (101)_2$. Then the key looks like:

$$(110010110)_2 = (626)_8 = (406)_{10}.$$

.

4.1.2 Mapping from 1D to mD

The Morton-quanta are independent for every level (see Example 4.0.4) and the structure is known (see previous section). Construct the quanta for every level and place the proper bit in the partial key. An example is shown in 4.1.2 and a FORTRAN90 implementation can be found in Appendix A.2.

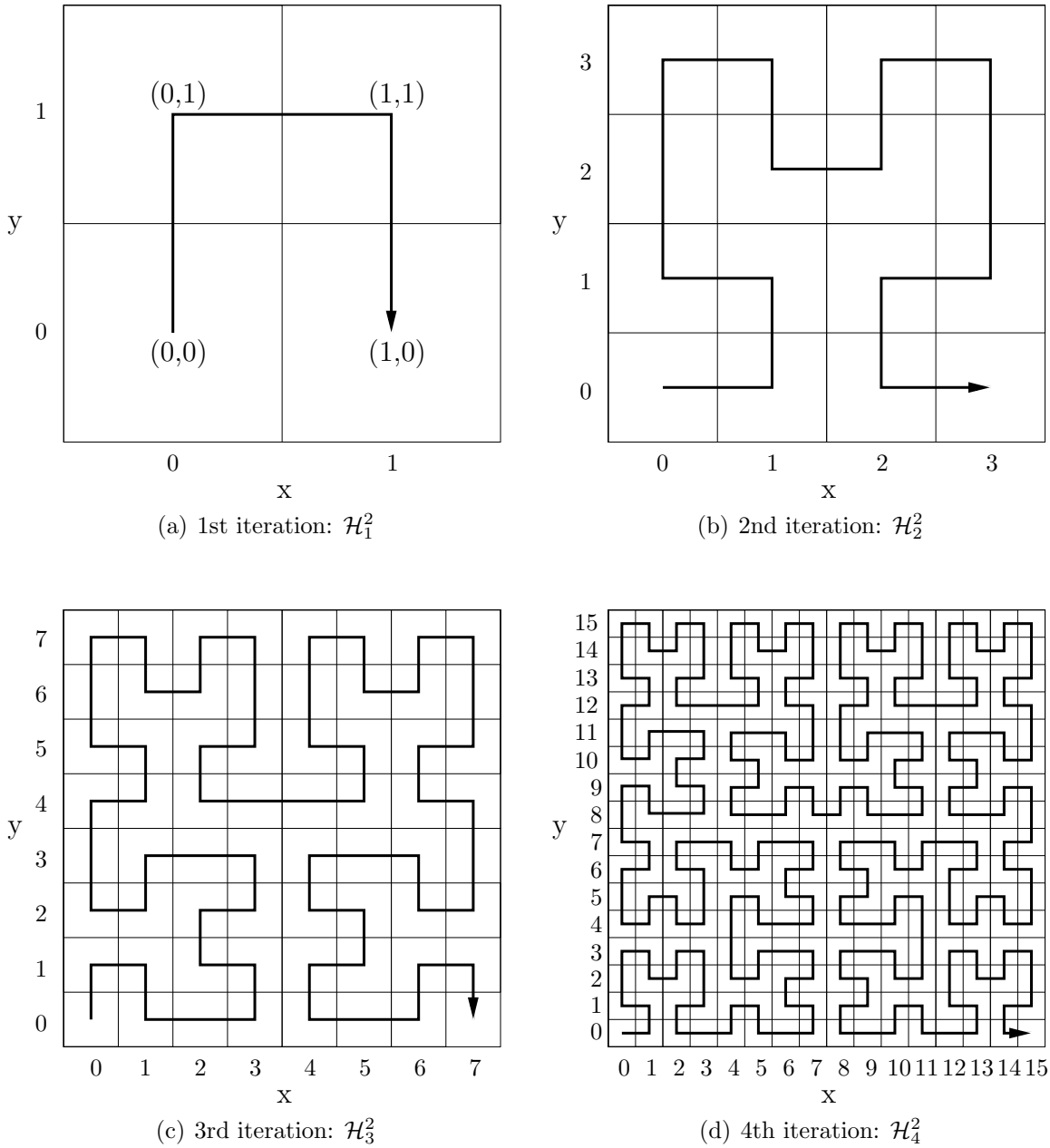
Example 4.1.2. We take the Morton-derived key $(110010110)_2 = (626)_8 = (406)_{10}$ of the previous Example 4.1.1. Starting with 1st level bit triple. The value is $(110)_2$. Then the partial keys are: $i_x = (0XX)_2$,$i_y = (1XX)_2$ and $i_z = (1XX)_2$. The upper case X is a placeholder for unset bits. For the 2nd level the bit tripel is $(010)_2$. Hence, the partial keys have the values: $i_x = (00X)_2$,$i_y = (11X)_2$ and $i_z = (10X)_2$. In the last step the procedure is performed analogously with the final result: $i_x = (000)_2$,$i_y = (111)_2$ and $i_z = (101)_2$. This matches the input data for the inverse mapping.

4.2 The Hilbert-curve

In 1891 David Hilbert gave an example of a SFC. The so called Hilbert-curve achieves the best data locality also known as locality preserving. For further reading [22] is listed to get insights in the metric for the rating. Locality preserving means, that the locality between objects in the multi dimensional space is approximately preserved in the linear space. In Figure 4.2, the first discrete realisations $\mathcal{H}_1^2, \ldots, \mathcal{H}_4^2$ are shown.

The first byte-orientated algorithm was presented in 1971 by Arthur Butz [74]. In contrast to the Morton-curve, the generation is not so simple. Whereas the Morton-curve is composed out of many z-shaped sub curves, the Hilbert-curve combines many u-shaped patterns with a different orientation. The algorithm of Butz calculates the rotations and reflections by complex bit operations dynamically and has restricted his approach to the 2D case. As a modification for higher dimensions would be too complicated, it is not used in the 3D BH tree code PEPC.

A table driven method for Hilbert-curves in various dimensions is presented in [75]. This method has to analyze the Hilbert-curve dynamically and its complexity is much higher than the Morton-curve mapping.



(a) 1st iteration: $\mathcal{H}_1^2$

(b) 2nd iteration: $\mathcal{H}_2^2$

(c) 3rd iteration: $\mathcal{H}_3^2$

(d) 4th iteration: $\mathcal{H}_4^2$

Figure 4.2: Geometrical construction of the $\mathcal{H}$-curve in 2D. The first 4 iterations $\mathcal{H}_1^2, \ldots, \mathcal{H}_4^2$ are shown. According to the level of the curve $1, \ldots, 4$ the partial keys $i_y = i_x = 0, \ldots, 2^{\texttt{level}} - 1$ are marked.

4.2.1 The Fast m-Dimensional Hilbert Mapping Algorithm

A more efficient general algorithm ($\mathcal{O}(\texttt{nlev})$) is the one advised in [76, 77]. This method is based on a static evolvement rule table and is called the FHMA (Fast Hilbert-Mapping Algorithm). This information must be generated once. The algorithm makes use of the fact, that the Hilbert-curve consists of many rotated and mirrored 1st order Hilbert curves H_1^m. Out of the original upside-down u-shaped pattern various derivates can be built by transformation. This transformations can be transcribed by coordinate exchanges and reverse operations.

There are two essential steps. Firstly, the initial state of the curve is defined regarding to the Morton-derived quanta. This is shown in Example 4.2.1 for 2D. With the help of that specification, the static evolvement rule table is built by the comparison of the entry and exit of the 1st and 2nd order Hilbert-curve.

The initial state of the curve must be assembled, the so called m-dimensional Hilbert-cell C^m. This data structure depicts the path of the 1st order curve in mD. The input parameter is the Morton-quantum τ and $C^m(\tau)$ maps it to the Hilbert-quantum:

$$C^m : \{0, \ldots, 2^m - 1\} \ni \tau \to C^m(\tau) \in \{0, \ldots, 2^m - 1\}\,. \tag{4.4}$$

Example 4.2.1. Exemplary this is done for the 2D case. Overall the 1st-level curve passes through 4 points (see Figure 4.2(a)). The Hilbert-curve starts at $(x, y) = (0, 0)$ with key 0. Therefore $C^2((yx)_2) = C^2((00)_2) = 0$. Then the curve passes the cell $(0, 1)$. So $C^2((10)_2) = 1$. After that the curve reaches the point $(1, 1)$. In the same way $C^2((11)_2) = 2$ is constructed. Finally, the curve ends at $(1, 0)$. For the Hilbert-cell it means that $C^2((01)_2) = 3$.

The Hilbert-cells for the most common dimensions $C^2(\tau)$ and $C^3(\tau)$ are listed in the following. The notation can be interpreted as an array, with the index τ. E.g. $C^2(2)$ maps on the value $(11)_2$.

$$C^2(\tau) = \begin{pmatrix}(00)_2 & (10)_2 & (11)_2 & (01)_2\end{pmatrix}^\mathsf{T}$$

$$C^3(\tau) = \begin{pmatrix}(000)_2 & (001)_2 & (011)_2 & (010)_2 & (111)_2 & (110)_2 & (100)_2 & (101)_2\end{pmatrix}^\mathsf{T}$$

With that information the evolvement rule table G^m, the so called mD Hilbert-gene, can be built:

$$G^m : \{0, \ldots, 2^m - 1\} \ni \tau \to G^m(\tau) \in \{0, \ldots, 2^m - 1\}^2\,. \tag{4.5}$$

This two column matrix (array) saves coordinate transformations, means exchange (1st column) and reverse operations (2nd column), that must be applied for every quantum.

The value is stored in binary representation. The index τ is the Hilbert-quantum. The value can be interpreted as follows: if a bit is set, then this coordinate is selected. The sequence of the coordiantes is $(x_m, \ldots, x_1)$. Primarily we concentrate on the exchanges in the 1st column, that are rotations in a geometrical context. The rotation axis is the first angle bisector. To demonstrate the usage of $G^m(\tau)$ a little example is composed.

Example 4.2.2. We take the 3D case. Like the Morton-order tripel the Hilbert-cell `horder`$=C^3((zyx))$ is used as the index for the evolvement rule table: G^3(`horder`). For example the Morton-triple of $(5)_{10} = (101)_2$ provides $(110)_2$, the 6th octant of the Hilbert-curve. This is put in $G^3[(110)_2][0]$ and the 1st column value is $(6)_{10} = (110)_2$. Hence, z and y must be interchanged, because the structure of the value of G^3 is (zyx).

The 2nd column of $G^3[0, \ldots, 2^3 - 1][1]$ depicts reverse operations, equivalent to reflections. Analogously, the value $(6)_{10} = (110)$ means, that z and y, $(5)_{10} = (101)$ embodys that z and x must be reversed bitwise.

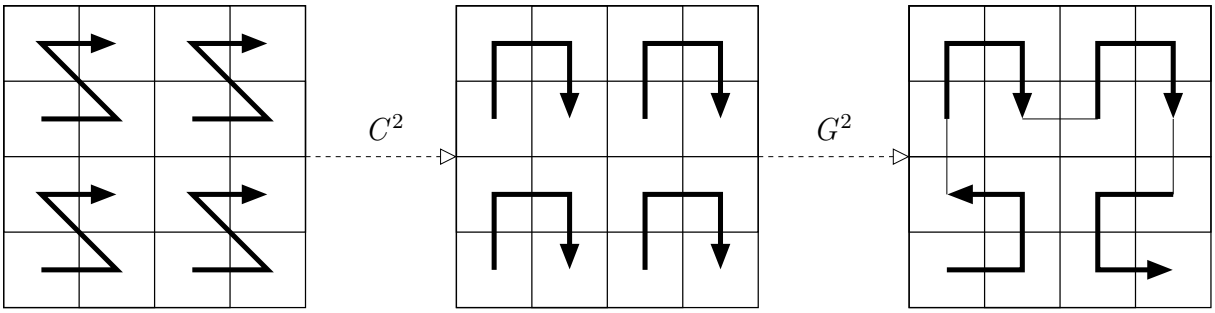


Figure 4.3: Basic idea of the Fast m-dimensional Hilbert-mapping algorithm shown for a 2D example.

In [76] an algorithm for the m-dimensional Hilbert-gene G^m is presented. All gene information are calculated with the knowledge of C^m by very complex bit operations. Nevertheless, the Hilbert-gene can be constructed geometrically, but for high dimensional quantized spaces, there exists the necessity of calculation rules.

Now the additional memory requirements of the FHMA are described. In comparison to the Morton-mapping, the memory footprint is larger, since the transformations commands must be stored. For a general dimension, $3(2^m - 1)$ values are needed in C^m and G^m. We take the 3D case, where every value can be expressed by a 8 bit integer, then 21 values, 168 bit or 21 Byte are neccessary. For higher dimensions the memory footprint grows exponentially, but for PEPC the 21 Byte (3D) are not measurable.

The trick of this algorithm is depicted in Figure 4.3. Firstly, the Morton-triple is transformed by C^2 to the u-shaped Hilbert-pattern. These patterns are incorrectly orientated and a combination to a Hilbert-curve would produce nonsense. Thus, with G^2 the patterns

are transformed and can be combined to the 2nd-order Hilbert-curve. For higher order curves (Higher level) the coordinates are transformed each step. In Example 4.2.3, the Hilbert-curve is geometrically generated for the 2D case. In Figure 4.4 the basic rotations and reflections are displayed, also for 2D.

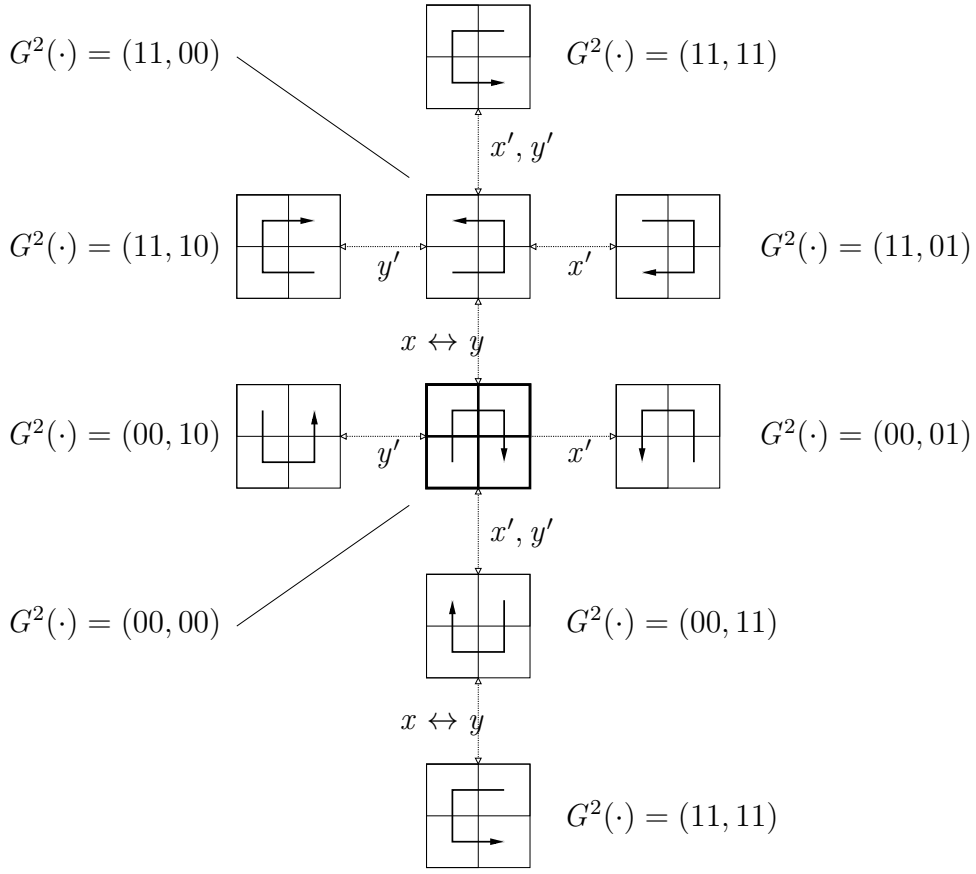


Figure 4.4: Rotations and Reflections for the Hilbert-cell in 2D. The original Hilbert-cell in the middle, as a basis for transformations, is highlighted. Furthermore, operations between the realisations are listed: $(\cdot)'$ denotes a bitwise reverse operation and $(\cdot) \leftrightarrow (\cdot)$ denotes an exchange operation. Additionally the evolvement rule table G^2 is assembled for every realisation of the Hilbert-cell. The acronym $G^2(\cdot) = (\mathrm{A}, \mathrm{B})$ means, that the 1st column, the exchange operation, has the value A and the 2nd column, the reverse operation, has the value B. Both, A and B are displayed in binary representation.

Example 4.2.3. We take a look at the 2nd level Hilbert-curve in Figure 4.2(a). This curve is composed out of rotated and reflected 1st level curves. In the 0th quadrant, the curve is rotated by the first angle bisector. So x and y are changed and the evolvement rule table has the value $G^2[0][0] = (11)_2$. In the 1st and 2nd quadrant the partial Hilbert-curve

equals the original upside-down u-shaped pattern. Therefore, not any transformation must be applied and G^2 is zero for both quadrants and columns. In the 3rd quadrant the curve is reflected and rotated alike, so x and y are exchanged and reversed bitwise. For the rule table it means: $G^2[3][0] = (11)_2$ and $G^2[3][1] = (11)_2$.

Overall following transformations arise for the 2D Hilbert-curve:

$$G^2 = \begin{pmatrix} (11)_2 & (00)_2 & (00)_2 & (11)_2 \\ (00)_2 & (00)_2 & (00)_2 & (11)_2 \end{pmatrix}^{\mathsf{T}}. \tag{4.6}$$

For 3D, in turn geometrical considerations can be applied, but a detailed description would be too complex. Here, the algorithm from [76] is used and the Hilbert-gene can be seen in the following:

$$G^3 = \begin{pmatrix} (101)_2 & (110)_2 & (000)_2 & (101)_2 & (101)_2 & (000)_2 & (110)_2 & (101)_2 \\ (000)_2 & (000)_2 & (000)_2 & (101)_2 & (000)_2 & (000)_2 & (110)_2 & (101)_2 \end{pmatrix}^{\mathsf{T}}. \tag{4.7}$$

4.2.1.1 Inverse Mapping from mD to 1D

Now the preliminaries C^m and G^m for the FHMA are available. The algorithm starts with the curve of the coarsest resolution $\mathcal{H}_1^m$, that leads across 2^m quanta containing $2^{m\cdot\texttt{nlev}}/2^m$ elements each (8 octants in 3D containing $2^{3\cdot\texttt{nlev}}/8$ points each). Now the Morton-quantum is formed from the highest available bits (at position $\texttt{nlev}-1$) of the partial keys. Suitable to this, using the vector C^m, the quantum of the Hilbert-curve is determined. This bits are inserted into the key at the highest position (bits from $m\cdot\texttt{nlev}-1,\ldots,m\cdot\texttt{nlev}-m-1$). For the next step, the coordinates are transformed for a correspondency to a Hilbert-curve in the correct orientation. This is done by the Hilbert-gene G^m.

In the 2nd step, in turn the quantum of the Morton-curve is intended, this time from the 2nd highest available bits ($\texttt{nlev}-2$) of the partial keys. After that, in turn, the quanta of the Hilbert-cell, determined by C^m, is inserted into the key (bits from $m\cdot\texttt{nlev}-m-1,\ldots,m\cdot\texttt{nlev}-2m-1$) and the coordinates are transformed for the next step with G^m. The same task continues for all `nlev` steps.

A Fortran90 implementation is in Appendix A.3. Obviously, the complexity of the FHMA equals the bit interleaving of the Morton-curve, $\mathcal{O}(\texttt{nlev})$. In this case, additional operations, result from the transformation of the partial keys (if-queries) and the access to the Hilbert-quanta in C^m and transformations in G^m, are introduced. In Example 4.2.4 a 3D example is composed.

Example 4.2.4. We take the same 3D partial keys as in Example 4.1.1. The 1st level Morton-octant is $(110)_2$. The function C^3 delivers the 4th Hilbert-octant. This is applied

to the key: $(100XXXXXX)_2$. Then the curve is transformed for the next step. So G^3 provides the transformation commands for the 4th Hilbert-octant. That gives $G[4][0] = (101)_2$, means z and y must exchanged, and $G[4][1] = (000)_2$, so no coordinate must be flipped. After that, the transformations are performed on the coordinates. Now the algorithm is ready for the next step.

In the 2nd step the Morton-octant is $(010)_2$. We look in C^3 and the 3rd Hilbert-octant results. This Hilbert-octant is applied to the key: $(100011XXX)_2$. We look in the Hilbert-gene: $G[3][0] = (101)_2$ and $G[3][1] = (101)_2$. We perform the exchange of z and x and the bitwise reverse of z and x.

In the 3rd step, the Morton-triple is $(110)_2$ and $C^3[(110)_2] = (010)_2$. We apply this to the key: $(100011010)_2$.

4.2.1.2 Mapping from 1D to mD

It is the same procedure as for the Morton-curve, but the transformations must be reverted. Furthermore, attention must be paid, because in general the transformation commands are not commutative. Hence, the calling sequence must be reverted, to guarantee correct partial keys. This issue was not respected in [77].

As a structure for the speedup of the calculation which can be derived by Equation 4.4 the inverse Hilbert-cell is defined, that maps from the Hilbert-quanta to the Morton-quanta. Now the algorithm is described.

The 1st level Hilbert-quantum is gathered. An evaluation of $C^{m,-1}$ for the Hilbert-quantum gives the Morton-quantum. The structure of the Morton-quanta is known. Thus, it can be split in the single bits and applied to every coordinate, similar to the Morton-mapping from 1D to mD.

The trick of the algorithm can be seen in the 2nd step. The 2nd level quanta is gathered from the Hilbert-derived key. Now, we look in the Hilbert-gene G^m and get the coordinate transformations. We apply this coordinate transformations to the partial keys, then the coordinates are in correct orientation and the transformations of the inverse mapping are reverted. In turn the Hilbert-quantum is translated by C^m to the Morton-quantum, and the bits are attached to the transformed partial keys.

For all other steps it is continued in the same way until all levels are resolved. After that, the partial keys can be obtained. In Example 4.2.5 this procedure can be studied. A FORTRAN90 implementation can be found in Appendix A.4.

Example 4.2.5. For 3D the inverse Hilbert-cell, where τ denotes the Hilbert-quantum, is:

$$C^{3,-1}(\tau) = \begin{pmatrix}(000)_2 & (001)_2 & (011)_2 & (010)_2 & (110)_2 & (111)_2 & (101)_2 & (100)_2\end{pmatrix}^{\mathsf{T}}$$

We recall the Hilbert-derived key from Example 4.2.4: $(432)_8 = (100011010)_2$. We start with the coarsest refinement. The 1st level Hilbert-octant is $(2)_8 = (010)_2$. We look

in the inverse Hilbert-cell $C^{3,-1}$, then the Morton-quanta is $(011)_2$ and apply this to the coordinates: $i_x = XX1$, $i_y = XX1$ and $i_z = XX0$.

The 2nd level Hilbert-octant is gathered, e.g. $(3)_8 = (011)_2$. Now the transformations must be reverted and we look in the Hilbert-gene for the 3rd Hilbert-octant: $G^3[3][0] = (101)_2$, so z and x must be exchanged, and $G^3[3][1] = (101)_2$, so z and x must be bitwise reversed. We perform the reverse operation firstly, that gives: $i_x = XX0$, $i_y = XX1$ and $i_z = XX1$. Then the rotation is performed by an exchange operation: $i_x = XX1$, $i_y = XX1$ and $i_z = XX0$. We look in $C^{3,-1}[(011)_2] = (010)_2$, which provides: $i_x = X01$, $i_y = X11$ and $i_z = X00$.

In the last step the Hilbert-octant $(4)_8 = (100)_2$ is gathered from the derived key. The last transformations must be applied. In this case $G^3[(100)_2][0] = (101)_2$. This is equivalent to the exchange of z and x. The last exchange gives: $i_x = X00$, $i_y = X11$ and $i_z = X01$. The Hilbert-cell gives the 6th Morton-octant. For the partial keys it means: $i_x = 000$, $i_y = 111$ and $i_z = 101$. These partial keys are matching the input for the inverse mapping in 4.2.4.

Remark 4.2.6. The Fast m-dimensional Hilbert-mapping algorithm analogously works for the Morton-curve, if the structures are adapted. The matrix G^m is overall zero and C^m is the identical mapping.

4.2.2 Patterns of the Hilbert-curve in 2D

As mentioned, Eliakim Hastings Moore introduced a space-filling curve in 1900 [66]. This curve is only a pattern of the Hilbert-curve. The 1st level Moore-curve equals the curve that was introduced by Hilbert, whereas higher-order Moore-curves follow other paths, but are a compositon of u-shaped patterns, too. In Figure 4.5 the different patterns are constructed for the 2D case geometrically. It is started with the 1st quadrant. Here the entry and exit of the partial curve are decisive. The 1st level path for every alternative pattern is derived from the original curve, but the single ingredients (partial curves) are differently orientated.

The Figure 4.5 is evaluated, where the number of patterns for any quadrant in the 2D case is derived, in the following. Overall there are 8 possible realisations of the 1st level Hilbert-cell in 2D. This number comes from the multiplication of all possible transformations in the Hilbert-gene $(((00)_2, (01)_2, (10)_2, (11)_2) \times ((00)_2, (01)_2, (10)_2, (11)_2))$. This number is halved since commutativity holds for some combination of rotations and reflections (see Figure 4.4). Taking all possible rotations, then the reflection with the same input parameter τ produces the same pattern. Additionally, a combination of any rotation with the "reflection" $(00)_2$ is commutative. Accordingly, the overall number of $4 \cdot 2 = 8$ results.

In the next construction step, the entry and exit is confirmed for the 2nd quadrant. This is depicted in Figure 4.6. Here only curves with the correct direction are taken, that

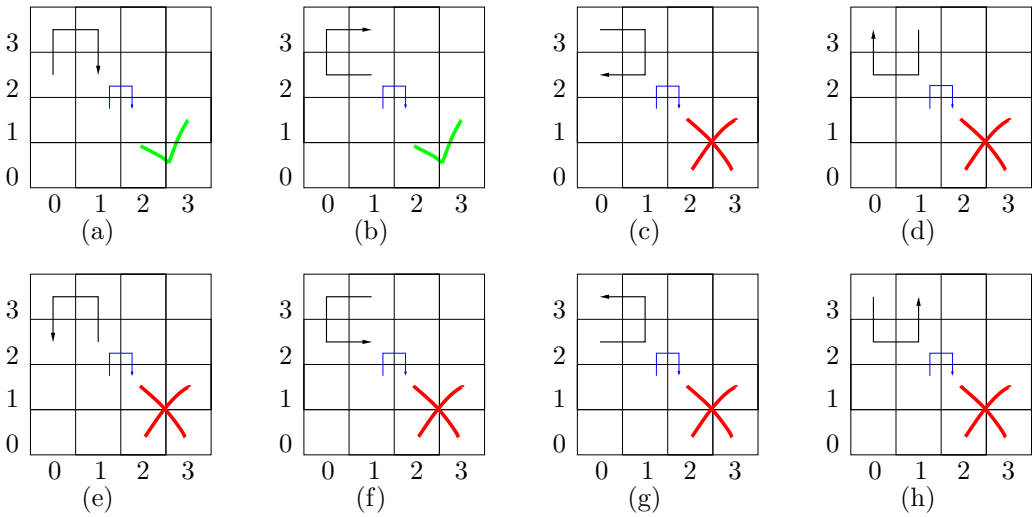


Figure 4.5: Adequate Hilbert-realisations for the 1st quadrant. Realisation that cannot match the Hilbert-cell are marked with a red cross. The path of the curve is marked with a blue curve in the center of any figure.

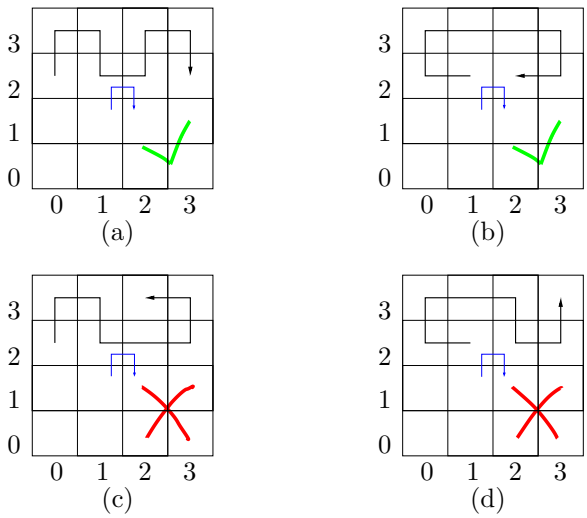


Figure 4.6: Confirming the entry and exit of the Hilbert-pattern for the 2nd quadrant. Likewise, adequate solutions are marked with a green hook.

obey a correct path of the 2nd level Hilbert-curve. Thus 2 possibilities of the path of the Hilbert-curve arise for the combination of the 1st and 2nd quadrant.

For the pattern in Figure 4.6, the 0th and 3rd quadrant must be filled with a partial curve of level 1. Overall, there are 3 different partial curves that can confirm the entry and exit of the Hilbert-curve in Figure 4.6(a). Hence, there are 3 over 2, means $3 \cdot 2/2 \cdot 1 = 3$ possibilities of a combination, means 3 different patterns of the Hilbert-curve in Figure 4.6(a). The same yields for the pattern in Figure 4.6(b). So overall 6 different patterns arise, that are also described in [78] and can be seen in Figure 4.7.

For higher dimensional quantized spaces, other patterns of the Hilbert-curve exists. This is an issue for the future work. Firstly, the establishment of a general formula for the number of patterns in a specific dimension is required. Secondly, the algorithmic generation of the Hilbert-gene, because geometrical consideration are inconcievable. The Hilbert-genes, for all 2D patterns are listed in the following:

$$G_{\mathcal{M}} = \begin{pmatrix} (11)_2 & (11)_2 & (11)_2 & (11)_2 \\ (10)_2 & (10)_2 & (01)_2 & (01)_2 \end{pmatrix}^{\mathsf{T}}$$

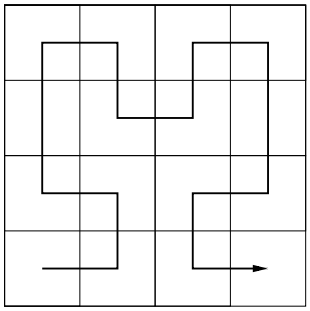
$$G^2_{\mathcal{L}_1} = \begin{pmatrix} (00)_2 & (00)_2 & (00)_2 & (00)_2 \\ (11)_2 & (00)_2 & (00)_2 & (11)_2 \end{pmatrix}^{\mathsf{T}}$$

$$G^2_{\mathcal{L}_2} = \begin{pmatrix} (00)_2 & (11)_2 & (11)_2 & (00)_2 \\ (10)_2 & (10)_2 & (01)_2 & (10)_2 \end{pmatrix}^{\mathsf{T}}$$

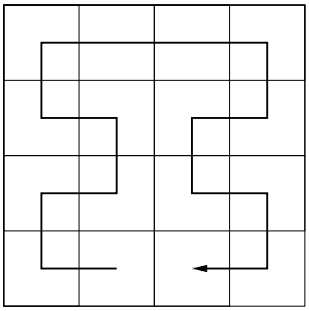
$$G^2_{\mathcal{L}_3} = \begin{pmatrix} (11)_2 & (00)_2 & (00)_2 & (00)_2 \\ (00)_2 & (00)_2 & (00)_2 & (11)_2 \end{pmatrix}^{\mathsf{T}}$$

$$G^2_{\mathcal{L}_4} = \begin{pmatrix} (00)_2 & (11)_2 & (11)_2 & (11)_2 \\ (10)_2 & (10)_2 & (01)_2 & (01)_2 \end{pmatrix}^{\mathsf{T}}.$$

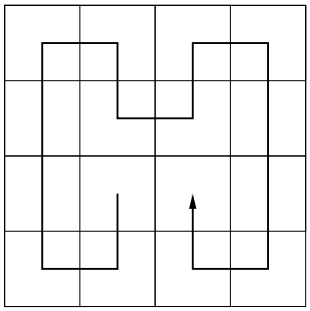
The simple application of the FHMA is not sufficient for all patterns and only works for Hilbert's original pattern. Therefore, some small changes are applied to the algorithm, to allow a fast generation of general Hilbert-patterns.



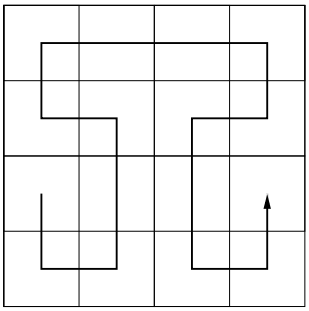
(a) Hilbert's pattern: $\mathcal{H}$



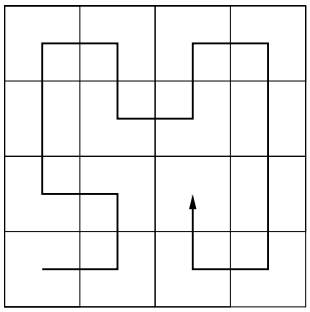
(b) Moore's pattern: $\mathcal{M}$



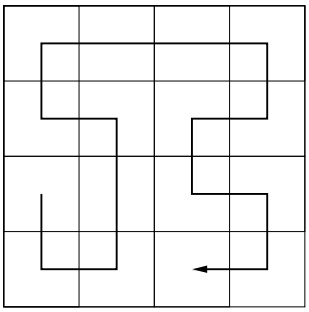
(c) The $\mathcal{L}_1$ pattern



(d) The $\mathcal{L}_2$ pattern



(e) The $\mathcal{L}_3$ pattern

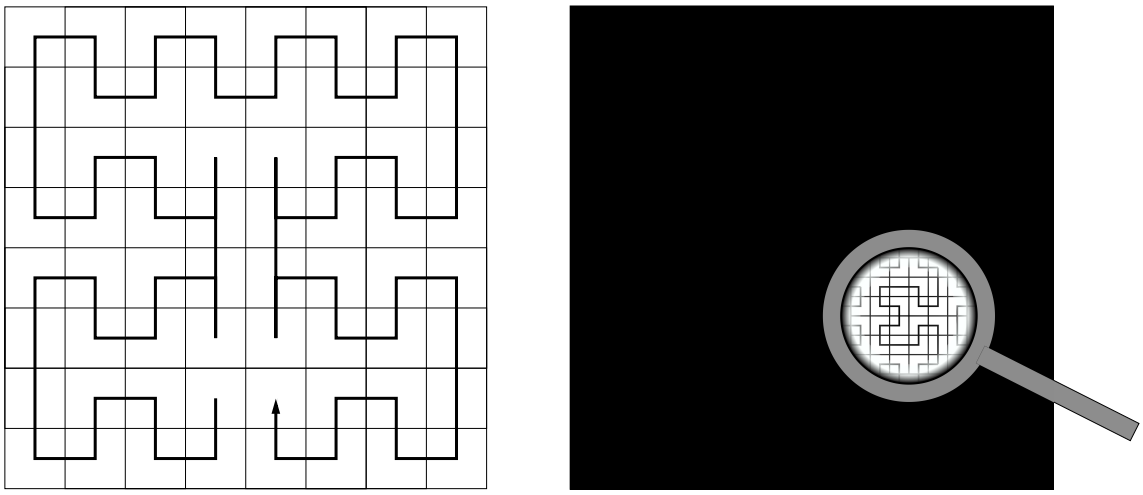


(f) The $\mathcal{L}_4$ pattern

Figure 4.7: The 6 different patterns of the Hilbert-curve in 2D. (a) Hilberts original pattern introduced in 1891. (b) Moores pattern, presented in 1900. (c)-(f) The 4 patterns of Liu [78] from 2004.

4.2.3 The Generalized Fast m-Dimensional Hilbert Mapping Algorithm

Without loss of generality we concentrate on the Moore-curve. As a motivation for the generalisation of the FHMA to all possible Hilbert-patterns, the output of the FHMA is shown in Figure 4.8(a). The generalized algorithm is abbreviated with GFHMA (Generalized FHMA).



(a) (b)

Figure 4.8: Motivation for the GFHMA. (a) The Moore-curve is generated by the FHMA. The combination of the different quadrants lacks. (b) Zoom in a Hilbert-curve of a high refinement level with a magnifying glass. Every Hilbert-pattern is composed out of many rotated and reflected 2nd order Hilbert-curves, of the original pattern. That is the basis for the GFHMA.

Based on the insight that every pattern of the Hilbert-curve is not only a composition of many 1st level, but also of fewer 2nd level Hilbert-curves, the idea for generalisation arises. This issue is illustrated in Figure 4.8(b). The FHMA would map to a composition of many Moore-patterns $\mathcal{M}_2^2$, that cannot be combined properly. The FHMA transforms the coordinates. The trick of the GFHMA is to omit the last two steps and to map the transformed coordinates on the original 2nd level Hilbert-pattern. Hence, an additional mapping is defined. This maps from the 2nd order $\mathcal{Z}$-curve to $\mathcal{H}_2^2$. This function is called the 2nd order Hilbert-cell $C_2^2(\tau)$, with the input $\tau = (y_1 x_1 y_0 x_0)$ in $2D$:

$$C_2^2(\tau) = \begin{pmatrix} 0 & 1 & 3 & 2 & 8 & 10 & 11 & 9 & 12 & 14 & 15 & 13 & 7 & 6 & 4 & 5 \end{pmatrix}^{\mathsf{T}}$$

For higher dimensions, an algorithm is required, which generates this structure. This is an issue for further work. In Figure 4.9, the general idea of the GFHMA is displayed.

We come back to the exemplary Moore-curve. In contrast to the curve in Figure 4.8(a) the application of the GFHMA delivers the correct Moore-curve, that is shown in Fig-

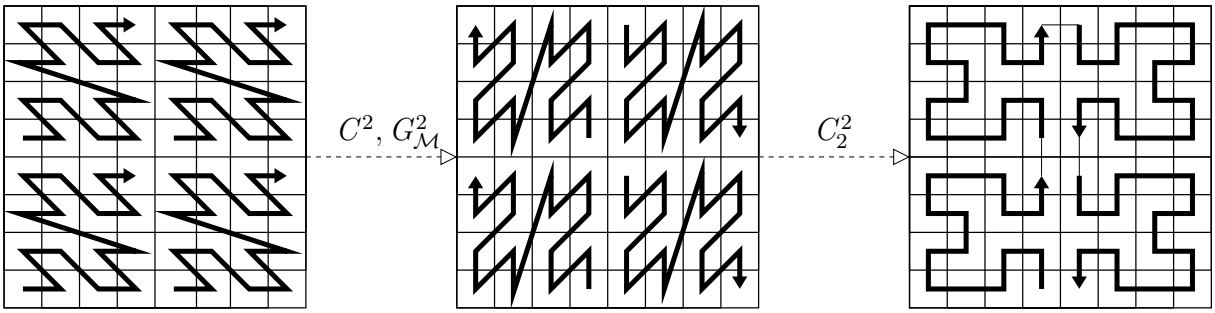


Figure 4.9: General idea of the GFHMA. The coordinates of the Morton-curve are the basis. Then this coordinates are transformed by the Hilbert-gene of the FHMA. After that, the rotated coordinates are mapped on a Hilbert-curve H^2_2 that are combined to M^2_3.

ure 4.10. This procedure works for all Hilbert-patterns alike, including the original one. The algorithmic complexity for a single key generation is of the order $\mathcal{O}(\texttt{nlev} - 2)$. This is fewer than the FHMA complexity, but more memory is needed.

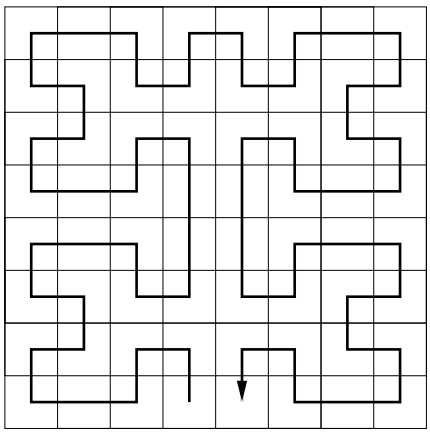


Figure 4.10: Generalized Fast Hilbert-mapping algorithm applied to the Moore-curve.

Remark 4.2.7. It is possible to map onto every SFC with $\mathcal{O}(1)$ with an array that maps the coordinates onto the derived key. The disadvantage is, that $2^{m \cdot \texttt{nlev}}$ memory locations are needed. For $m = 3$ and $\texttt{nlev} = 20$ this are $8^{20} \approx 1 \cdot 10^{18}$ locations. In this example the derived-key is of the integer type with 8 bytes. The $\approx 9\ 000\ 000\ 000$ GB vastly cross the memory boundary of any computer in the world. Balanced approaches are possible. E.g. approaches which map on a 5th iteration curve. Then this curves are combined to a higher order curve (if self-similarity holds). Then the order became $\mathcal{O}(\texttt{nlev} - 5)$.

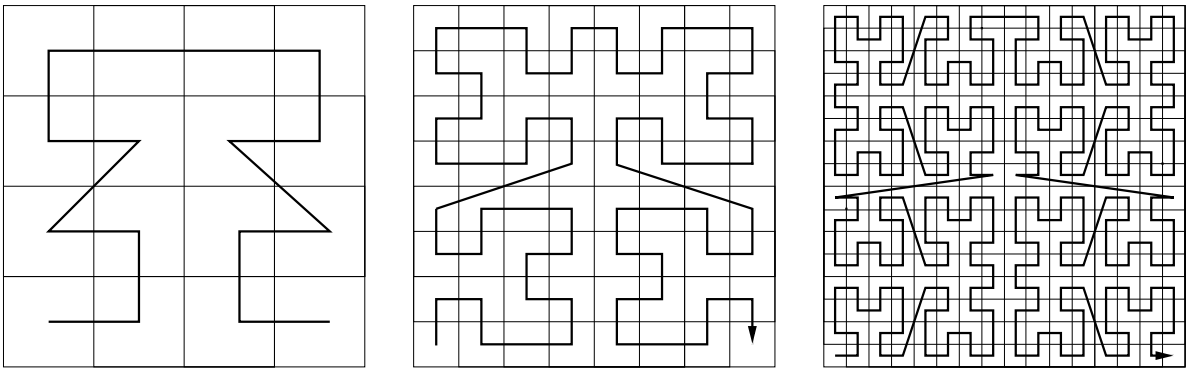
4.2.3.1 Definition of a Novel Space-Filling Curve

The GFHMA can also be used for fast mappings by novel space filling-curves. Here, a mixture of the Hilbert- and Morton-curve is established in 2D. The rule table $G^2_{3\mathrm{E}}$ for that

curve is shown in Equation 4.8, which is derived by Figure 4.4. The first iterations of the so called 3E-curve are displayed in Figure 4.11:

$$G^2_{3\mathrm{E}} = \begin{pmatrix} (11)_2 & (11)_2 & (11)_2 & (11)_2 \\ (00)_2 & (10)_2 & (01)_2 & (11)_2 \end{pmatrix}^{\mathsf{T}}.$$

This curve should demonstrate the versatility of the GFHMA and is only an example. Perhaps exactly this curve is well-suited for a specific application. In the following section the original Hilbert-pattern is compared with the Morton-curve for the parallel BH tree code PEPC.



(a) 1st iteration (b) 2nd iteration (c) 3rd iteration

Figure 4.11: The novel E3-curve is presented. This curve merges the Hilbert- and Morton-curve.

4.3 Effects of the Hilbert-Curve in PEPC

In this section, the effects of the Hilbert-curve and the reduction of algorithmic bottlenecks by the a-priori selection of a highly locality-preserving SFC, is studied. This issue is also mentioned in [12], but not examined in detail. The relevant steps of the code are picked out and the influence of the Hilbert-curve is discussed.

The calculation of Hilbert-derived keys is a little slower than the Morton mapping. The timings $t_{(\cdot)}$ refer to a single key calculation of the specific curve. Since in PEPC a maximal refinement level of `nlev` $= 21$ is implemented, the timings for the Hilbert-curve $\mathcal{H}^3_{21}$ and the Morton-curve $\mathcal{Z}^3_{21}$ are researched. The benchmarks of a single key construction were made with JUGENE and can be seen in the following:

$$t_{\mathcal{H}^3_{21}} = 8.9 \cdot 10^{-7} s > 6.1 \cdot 10^{-7} s = t_{\mathcal{Z}^3_{21}}.$$

The impact of this step for the overall run time of the code is minimal. E.g. for 500000 local particles, the additional amount of time is 0.14 seconds. This reflects the presumption, that the transformation commands retard the key calculation a little.

The influence for relevant steps of the HOT-scheme is explained in the following:

- Since the number of keys remains the same, the algorithmic complexity does not change for the parallel sorting step in the `domain decomposition`. Additionally, the increased data locality does not provide any measurable effect.

- No detectable improvement for the tree construction can be established by the Hilbert-curve. Since the branches are determined by the linear key space, only marginal changes take place through the different path of the Hilbert-curve. For an inhomogeneous setup, the branch number highly depends on whether the task limits are located within very dense regions or not. Hence the tree depth is very high and much branch nodes are the consequence (at least with the original branch definition). Therefore, it is random, just which path is cut in a dense region, and which curve is better regarding the `exchange` overhead.

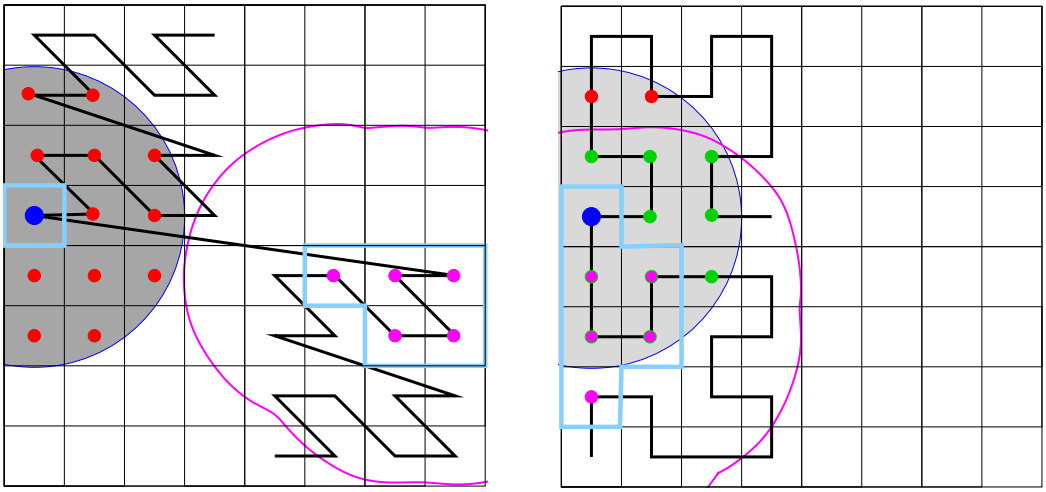
- Because the construction of global fill nodes highly depends on the previous steps, which have experienced only marginal changes, no relevant improvement is detected.

Finally, the Hilbert curve has a vanishing influence on the `domain decomposition` and the local and global tree construction.

We concentrate on the effect for the `tree traversal` and the `force summation`. The advantage of a locality-preserving curve is shown in Figure 4.12 and dicussed in the following. The Morton-curve jumps between the quadrants (2D). This can result in divided local domains that belong together in linear but not in mD space. If the particles are spatially near, then with a higher probability, non-local information can be shared that was requested by other local particles during the `tree traversal`. It is like the cache, that holds data that was used, because it assumes that it is needed in close-by future. In the case of the Hilbert-curve, the domains are always connected. The improved data-locality can optimize this issue. To simplify the matter for further reasons, it is assumed that each particle requests information in a certain halo.

In the following the selection of the setup is explained. For tasks numbers that correspond to a multiple of the number 8, the number of divided domain is intrinsically minimised. On the one hand, the local domains are aligned to power of 8 limits. This fact reduces the jumps of the Morton-curve. On the other hand, load-balancing ensures that the interactions are balanced. On that account, the number of divided domains decreases, since particles with a large interaction count are shifted to the adjacent task. This weaks the strength of the Hilbert-curve, regarding the communication. In an inhomogeneous setup indeed there could be divided domains. The problem is, that the probability of an isolated particle is lower then for the homogeneous setup. For this reason a homogeneous setup was simulated with 31 tasks. By this selection, the number of divided domains is maximised. Here, the first timestep is considered for empirical data, because load-balancing does not downsize the amount of divided domains. This has a simple reason. In the 1st time step the basic information for the load-balancing, the interaction count

of the previous step, is of course not yet available. Using this setup, the weakness of the Morton-curve is shown and the locality-preserving character of the Hilbert-curve is highlighted in Figure 4.13. In this figure the communication matrices for both curves, and in Table 4.1 statistical results, are shown. The tools SALSA [79] and VAMPIR [80], for the visualisation of communication, and SCALASCA [81], for statistical values, were used. Since the performance does not benefit from any debugging tool, only a small example was used. Here, 300000 particles were simulated with 31 MPI-tasks.



(a) Morton-curve (b) Hilbert-curve

Figure 4.12: Regarding to the requests of information in the `tree traversal` the advantage of a locality-preserving curve is shown. The neccessary information for the blue particle is simplified with a grey halo. Red circles mark the information that must be additionally requested for the blue particle. A magenta halo marks the information that was requested by the magenta particles and the light blue lines mark the local domain. In contrast to the divided domain in (a) the Hilbert-curve has a connected domain. For the Morton-curve many information must be requested, whereas the Hilbert-curve only needs two extra requests. Green particles, or particles with a magenta infill and green border, respectively, mark a information that is already locally available and was requested by the magenta particles.

As mentioned, the irregualar communication pattern of the `tree traversal` is the most interesting part for the analysis. Particularly interesting is, which task requests information from which other tasks. Therefore, the metric "number of messages" was selected. The information exchange in the `tree traversal` looks like this: the task needs non-local pseudo-particle information for the evaluation of the MAC. It knows the `key` and the `owner` by the hash table. Then it contacts the `owner`, which sends the requested information. Since the `owner` only sends the information of one node, the message size is constant. So, the communication matrix for the `tree traversal` is symmetric.

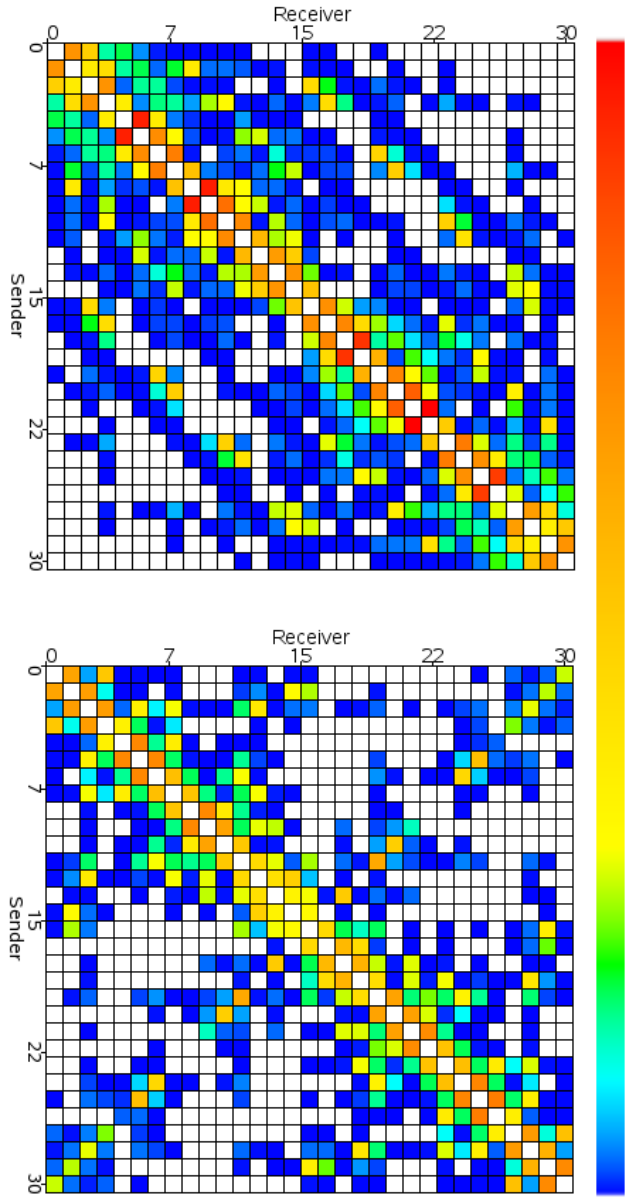


Figure 4.13: Comparison of the communication matrix for the Morton-curve (upper) and the Hilbert-curve (lower matrix) for a homegeneous setup that was simulated with 31 MPI-tasks. The number of messages is shown. The Hilbert-curve reduces the number significantly. On the one hand, this matrix is more sparse. On the other hand, the colors induce, that less communication is needed.

At the current state of the code, each send operation only ships the information of one node. A new approach would be to decide directly whether more information is needed or not, and to ship more information about the requested part of the non-local tree directly. This would vary the amount of data per communication, and other measurements, like the bandwidth are of interest.

	Morton-curve	**Hilbert-curve**	**ratio**
Average	1586	1262	80%
Standard deviation	98	65	66%
Minimum	1390	1144	82%
Maximum	1774	1384	78%
Overall messages	49152	39136	80%
zero-entries	31%	46%	148%

Table 4.1: Statistical evaluation of the message number per task in the `tree traversal` for the Hilbert- and Morton-curve. A setup with 300000 particles was simulated across 31 tasks. For the minimum the zero-entries are not respected. The Hilbert-curve provides an upgrade for the irregular communication pattern in the traversal at least for the depicted setup.

In the following, the described setup is researched for the number of messages. In this example, the Hilbert curve reduces the total number of communications by 20%. In addition, the communication and thus the network utilization is more balanced, which is clearly shown by the standard deviation. A notable matter is, that the tasks with the maximal number of messages, in case of the Hilbert-curve is still below the task with the minimal communication of the Morton-curve. Apparently, the communication matrix of the Hilbert-curve is more sparse, which is underlined by nearly 50% of zero-entries. Hence, for the Hillbert-curve each task communicates with a smaller number of other tasks. Regarding the communication, in any case the Hilbert-curve is better than the previously installed Morton-curve.

In addition to these outstanding impacts on the communication structure, the new curve brings no temporal effects for the `tree traversal` and the `force summation`, as it was initially suspected. That issue underlines the excellent implementation of the traversal routine, because the communication is cleverly hidden behind the calculation.

One consequence is to setup a MPI communicator, which uses the excellent improvements of the Hilbert-curve, to get a performance increase, by an adapted network allocation of the target machine.

In any case, the presence of an alternative SFC is a great deal for Pepc. Unfortunately, this curve is not the hoped-for improvement for the further decrease of the run time but has a massive impact on the communication structure. In the nearby future the modification of the internal communication structure of the network is tested.

The detailed accomplishment, concerning the generation, and generalization of the FHMA to all possible Hilbert-patterns (in this thesis for the 2D case), promisses a benefit

for other application as well as the detailed overview of SFC for various topics in computing sciences.

Chapter 5

Virtual Local Domains

In Chapter 3, the `exchange` of the branch nodes was identified as the major bottleneck, for the memory footprint and scalability, of the parallel HOT-scheme and the anticipated BH tree code PEPC. In this chapter, a novel approach is presented, that minimises the pure parallel overhead caused by branch nodes and in this context the expense of the `exchange`. The new concept, so called VLD (Virtual Local Domains) provides a lot of advantages compared to the original concept. E.g. the a-priori determination of the branch nodes is described in detail later.

The structure of the chapter follows the developmental steps and leads to a new and versatile branch concept. The insight emerged, that the current branch nodes are suboptimal. Currently, some branch nodes are far too deep in the tree, that means on a high level. Then an attempt was made to reduce the local number of branches and thus their global amount. Here, the VLDs come into play. For this novel approach a non trivial equation must be solved. Without a fast algorithmic solution of the equation, the whole theory of VLD would not make sense. Hence, an explicit formula is introduced. Once the requisites for the concept of VLD were established, it was integrated into the old branch finding algorithm. After that a very tight estimation for the branch nodes was found. This estimation can predict the global number of branch nodes, before the tree structure is generally known. The branches can be determined directly after the `domain decomposition` a-priori with $\mathcal{O}(1)$. This was also adapted for the original branch concept. On the basis of the estimate a new algorithm has been developed. In further investigations, the weaknesses of the novel concept were considered. The VLD depends heavily on the `domain decomposition`, but this is also the case for the original concept. For all of the consideration it is important to know, that the determination of the branch nodes is a problem of the linear key space, generated by the mapping rule of an arbitrary self-similar space-filling curve. As mentioned in Chapter 4, the branches are independently of the SFC.

First of all, the weaknesses of the original branch concept are highlighted. At the limits of the local domain, the original concept causes many small branch nodes at a high level, which may be replaced by a single branch at a lower level. This effect is illustrated for 2

tasks in Figure 5.1. The problem can be recognized equally for both tasks.

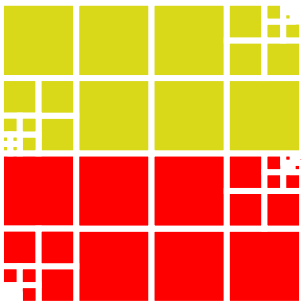


Figure 5.1: Small branch nodes at the edges of the local domains, that arise by the original branch concept for two tasks.

As a reminder, the keys, that were generated during the domain decomposition, are stored in the array `pekeys`. Thereby, the local domain of the task p can be expressed by $k_{p,min}$ = `pekeys(1)` and $k_{p,max}$ = `pekeys(np)`, where `np` describes the number of local particles. All local tree nodes, including the fill nodes are stored in array `treekeys`. The problem is caused by the exact resolution of the local domain. For this reason, very small branches are detected at the limit of the local domain. The current algorithm is not implemented perfectly, but this fact is discussed later. Since it is the goal to get as few branch nodes as possible, the algorithm starts at level 1, where the largest nodes are placed, regarding the contained `nlev`-cells. As a start condition, all keys with the level 1 are filtered from the array `treekeys`. The calculation of the level is very easy:

$$\texttt{level} = \log_8(\texttt{key}). \tag{5.1}$$

Then it is checked whether all nodes below this key are lying entirely in the local domain or not. Therefore, the parent cell of the boundary particles, $k_{p,min}$ and $k_{p,max}$, that applies to the current level, is encrypted. This can be done by a simple shift operation, as depicted in Example 5.2.1. In the following these parent cells are called `leftLimitMeLevel` and `rightLimitMeLevel`. Left and right is according to the linear space.

Finally, each tree node at the certain level, denoted with `searchkey`, is compared with the corresponding parent-cell of the boundary particle. The branch condition is:

$$\texttt{leftLimitMeLevel} < \texttt{searchkey} < \texttt{rightLimitMeLevel}. \tag{5.2}$$

If the condition is satisfied, then `searchkey` is included in the list of branch nodes. Otherwise, the key is split into its children. For the next level and a finer resolution the child keys are stored in a todo-list. For any level the todo-list is sorted. Finally, the algorithm terminates, when the todo-list for the next level does not contain any key. The disadvantages of this algorithm are:

- The level for approximately $8/7 \cdot N/P$ (homogeneous) or $2 \cdot N/P$ (inhomogeneous) nodes must be calculated for the initial condition (the number of overall nodes, means leaves plus twigs, was described in Section 3.1.2, page 25).
- The local tree must be traversed, in a worst case, up to `nlev`.
- The branch candidates in the todo-list are sorted for each level.
- The branch condition, which reflects the original concept.
- The todo-list length must be estimated and causes memory waste.

Since the tree nodes in `treekeys` are based on `pekeys`, in any case the twigs at the edge of the local domain are resolved at a finer level. This continues level-by-level until any twig at the edge is resolved in all its leaves, as illustrated in Figure 3.10, page 29, where almost all the leaves have been detected as a branch. This process is generally described in Figure 5.2. Here, an abstract view on the nodes, which are branch candidates, for each level is provided. The nodes at the edges (left and right) are divided into their child cells. This matter is described by a red colored cell, which is divided in 4 children (2D). On the next level the children are resolved. The process continues and in turn the nodes at the edges are not recognized as a branch and resolved in 4 (child) keys thereof.

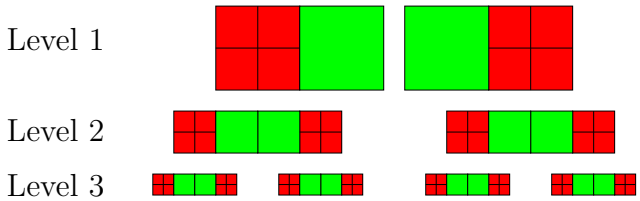


Figure 5.2: Process that leads to very fine branches at the edges of the local domain. Green cells mark a detected branch. Red cells at the edges are resolved in their children and marked red.

5.1 Idea of Virtual Local Domains

Currently, unused space exists between the local domains, as shown in Figure 5.3. The idea arises, that this space can be optimally shared between adjacent tasks in order to reduce the number all local branches. The small branches at a high level, get the possibility to jump to a lower level, because they get more space. Another and more important point is, that the possibility exists that several branches merge into a common parent cell into a single branch node. Accordingly, with the minimisation of all local branch nodes, the global amount is optimized.

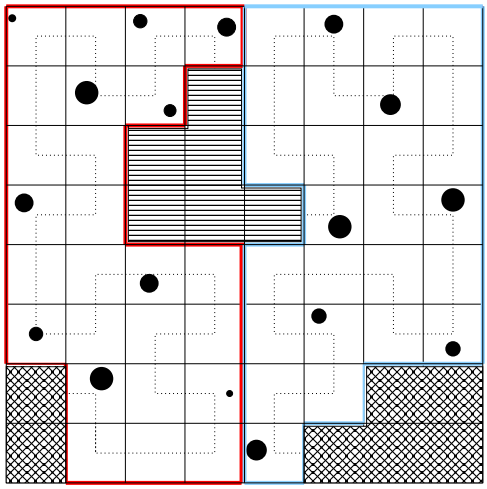


Figure 5.3: The unused key space is shown for 2 tasks. The domain of task 0 is red and the domain of task 1 is light-blue colored. It is differentiated between two types of unused space. Firstly, the space between two tasks (e.g. between task 1 and 2), that is filled with horizontal lines. Secondly, the space between the limit of the global key space and the first and last tasks, respectively. This is highlighted with crossed lines.

The unused space is bounded by exactly two tasks. Both should have a minimum number of local branches, therefore the range must be shared fairly, to minimize the global sum. The fact is exploited that branches are nothing else than tree nodes. Especially, the start of a branch is located at discrete keys (of the level `nlev`), see Definition 3.1.2 (iii), page 28.

The unused space between task p and task $p+1$ can be uniquely identified by the keys $k_{p,max}$ and $k_{p+1,min}$. The limits can be accessed without any communication, since the boundary keys for adjacent tasks are stored. Moreover, branches start and end at multiples of the power of 8: $c \cdot 8^{\texttt{nlev}-\texttt{level}} \equiv c \cdot 8^{\tau}$. In order to achieve as large branch nodes as possible, a multiple of the highest power of 8 is required inside the unused space. Since the number of levels is limited, this leads to Equation 5.3. Colloquially, the power τ_p, the highest power in the unused space between task p and $p+1$, is maximised for any multiple $c \in \mathbb{N}$, in order to minimise the amount of local branches:

$$\tau_p = \max\left\{ i \ \in \ \{0, \ldots, \texttt{nlev}\} : k_{p,max} < c \cdot 8^i \leq k_{p+1,min} \right\}, \text{for any } c \in \mathbb{N}. \tag{5.3}$$

Let τ_p be the optimal branch level, then

$$L_p \in \left\{ \left\lfloor \frac{(k_{p,max}+1)}{8^{\tau_p}} \right\rfloor 8^{\tau_p}, \ldots, \left\lfloor \frac{(k_{p+1,min})}{8^{\tau_p}} \right\rfloor 8^{\tau_p} \right\}.$$

This defines the VLLs (Virtual Local Limits) L_{p-1} and L_{p+1} for the task p, thus spanning its VLD. If the VLLs are processed across all tasks, then the entire global key space

is used and covered by branch nodes. For the first and last task an extra rule is defined in a natural way. They can access the smallest or the largest key, as left respectively right VLL. In the following, the strength of the new concept is shown for a 2D example.

Example 5.1.1. We take a 2D example with two tasks (Hashed-Quad-Tree with basis 4), that corresponds to Figure 5.3, page 66. The domain of task 0 is limited by $k_{0,min} = 2$ and $k_{0,max} = 26$. For task 1 the limits are $k_{1,min} = 33$ and $k_{1,max} = 56$. The VLL between both tasks is $L_0 = 32$. Only cells that contain at least one particle are counted as a branch node. With the new approach, it is possible to reduce the number of branches from 16 to 4.

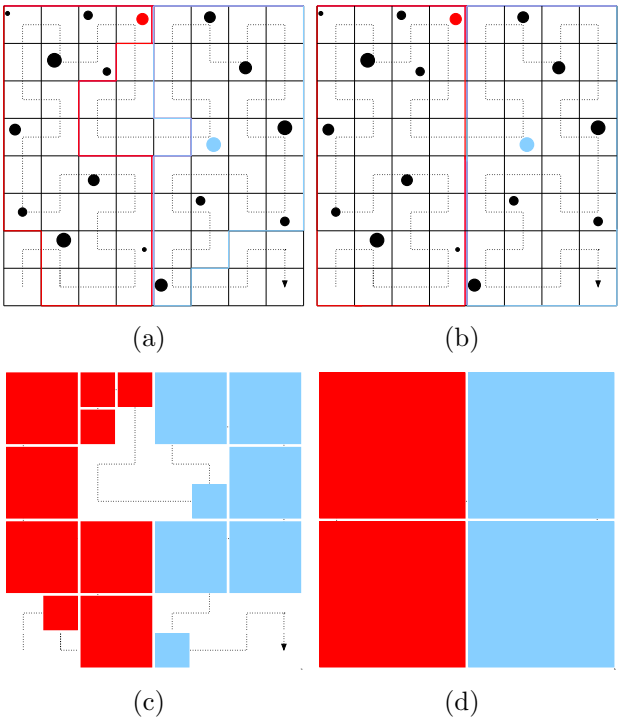
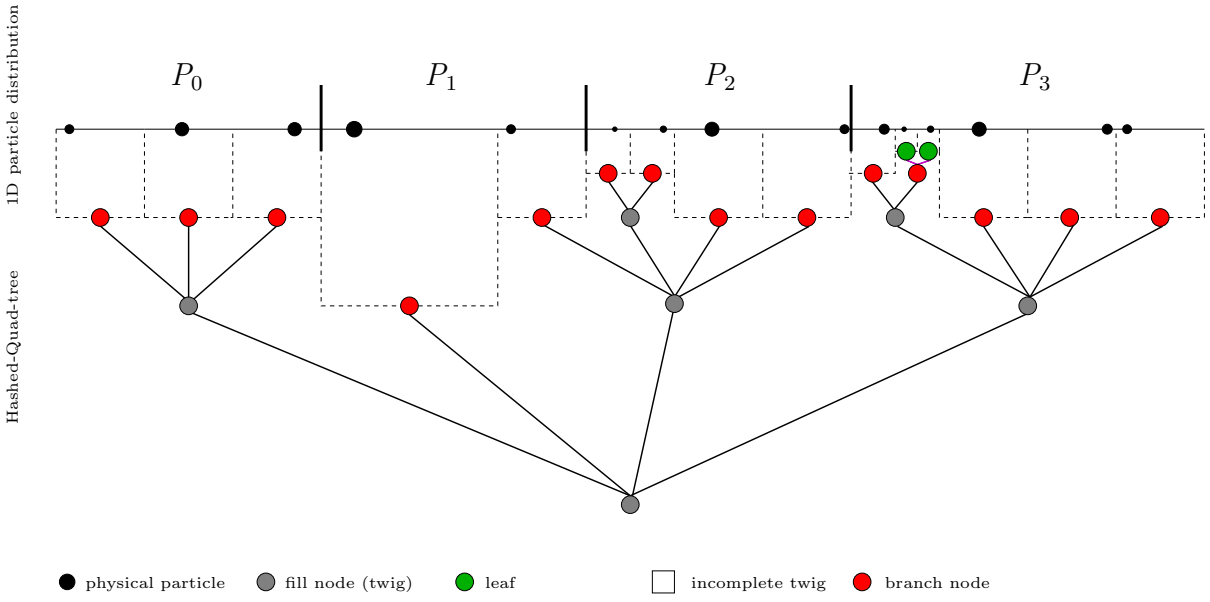
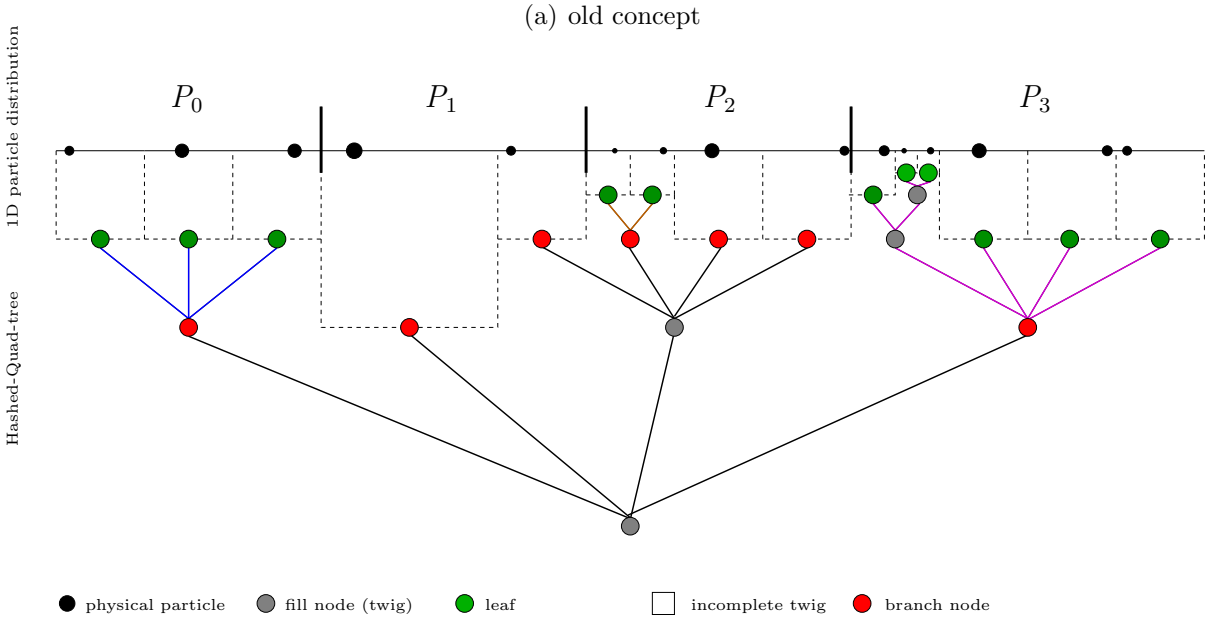


Figure 5.4: Comparison of the original and the Virtual Local Domain branch concept for 2 tasks and a 2D particle distribution. (a)-(b) The domains for both branch concepts are shown. The domain of task 0 is colored red and the one of task 1 is marked with light-blue. The boundary particles of every task are colored equally. In (b), the concept of VLD was used to adapt the domain. In (c)-(d), the branches are shown for both concepts. The novel concept provides a magnitudes smaller amount of branch nodes.

In Chapter 3 the flow of the HOT-scheme was depicted. At this stage, a hierarchical view of the data structure was shown. Supplementary, the strength of VLD is underlined by a comparison to the original concept, as illustrated in Figure 5.5, page 68. In this example, the global amount of branch nodes was halved from 14 to 7.



(a) old concept



(b) VLD

Figure 5.5: Comparison of the original branch concept and the novel Virtual Local Domains for a hierarchical view of a global tree. The example was also used in Chapter 3. The global amount was halved by VLD. For task 0, the local amount was reduced from 3 to 1 branch node. Here 3 children melt into one parent cell. For task 1, both concepts are equal. The number of branch nodes for task 2 is reduced by 1 and for task 3 a reduction from 5 to 1 can be observed.

As a necessary requirement for VLD, Equation 5.3 must be solved efficiently, to bring this concept into practice.

The highest power within an interval can be determined by the highest digit of the limits that differ. The $\textsc{Xor}(k_{p,max}, k_{p+1,min})$ highlights the different bits of $k_{p,max}$ and $k_{p+1,min}$. The application of a logarithmic operation provides the level, and therefore the highest power of 8 in the unused space (see Equation 5.1). Accordingly, the following applies:

$$\tau_p = \lfloor \log_8(\textsc{Xor}(k_{p,max}, k_{p+1,min})) \rfloor \,. \tag{5.4}$$

Example 5.1.2. We consider the interval $(33, 38]$ in a 2D example with basis 4:

$$\begin{aligned} (33)_{10} &= (100001)_2 \\ (38)_{10} &= (100110)_2 \\ \textsc{Xor}(33,38) &= (000111)_2 = (7)_{10}. \end{aligned}$$

So, $\tau_p = \lfloor \log_4(7) \rfloor$ is 1 and $L_p = \left\lfloor \frac{38}{4^{(1)}} \right\rfloor \cdot 4^{(1)} = 9 \cdot 4 = 36$ holds.

5.2 Application of VLD to the Branching Algorithm

The modified algorithm proceeds similar to the present one. However, the branch condition (see Equation 5.2) must be adjusted to VLD and special issues must be considered. Additionally, disadvantages of the current version are tuned. The keys from the array `pekeys` correspond to a quantisation of the level `nlev`. These keys, $k_{p,max}$ and $k_{p+1,min}$, are used to determine the VLL. So the VLL is also at this level. The keys from the field `treekeys` correspond to tree nodes at any level. To compare the keys at the same level, the appropriate information must be extracted from the VLL. This corresponds to the parent cell of the VLL of the specific level (the step) of the branch finding routine. This is determined by a shift operation, as explained in Example 5.2.1. This procedure is used in the same way for many other situations, like the `local tree` build.

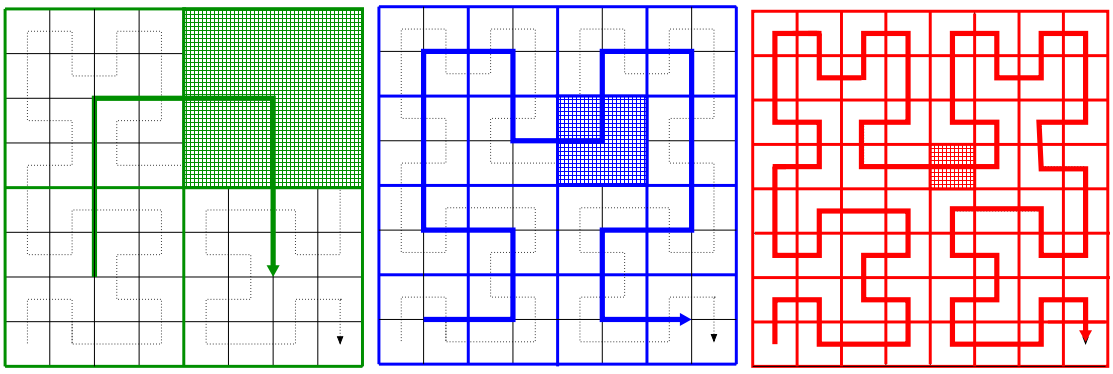
Example 5.2.1. The 2D case is considered with a maximal refinement level `nlev`=3. The limits of the unused space are $k_{p,max} = 26$ and $k_{p+1,min} = 33$. The VLL $L_p = (32)_{10} = (100000)_2$ is calculated by Equation 5.4. For every level $\texttt{level} = 1, \ldots, 3$ the parent cell is calculated. In binary representation, the bits of the VLL are described with $\lambda_j,\ j = 0, \ldots, 2 \cdot \texttt{nlev} - 1$:

$$\text{Level 1}: \quad (\underline{10}0000)_2 \Rightarrow \sum_{j=4}^{5} \lambda_j \cdot 2^j = (10)_2 = (2)_4$$

$$\text{Level 2}: \quad (\underline{1000}00)_2 \Rightarrow \sum_{j=2}^{5} \lambda_j \cdot 2^j = (1000)_2 = (20)_4$$

$$\text{Level 3}: \quad (\underline{100000})_2 \Rightarrow \sum_{j=0}^{5} \lambda_j \cdot 4^j = (100000)_2 = (200)_4 \,.$$

In Figure 5.6 the particular cells of the VLL are displayed.



(a) 1st level parent cell (b) 2nd level parent cell (c) 3rd level parent cell

Figure 5.6: Calculation of parent cells for any self-similar space-filling curve. The Hilbert-curve is used. The appropriate curve for every level is shown.

Another issue is, that the left VLL of each task apart from task 0, must be shifted by one to the left. This can be studied in Example 5.2.2. With the same considerations, the right VLL of the task $P-1$ must be shifted by one to the right. For this case, attention must be paid, because of a binary overflow.

Example 5.2.2. The same preconditions as in Example 5.2.1 hold. In the following, the branch algorithm is mimicked for the initial VLL and the modified VLL, for the first key of task 1, $k_{1,min} = 33$. The initial VLL is $L_0 = 32$:

$$\begin{aligned}
&\text{Level 1}: \quad (32)_{10} = (\underline{2}00)_4 \Rightarrow 2, (33)_{10} = (\underline{2}01)_4 \Rightarrow 2, \text{not fulfilled } (2 < 2)\\
&\text{Level 2}: \quad (32)_{10} = (\underline{20}0)_4 \Rightarrow 8, (33)_{10} = (\underline{20}1)_4 \Rightarrow 8, \text{not fulfilled } (8 < 8)\\
&\text{Level 3}: \quad (32)_{10} = (\underline{200})_4 \Rightarrow 32, (33)_{10} = (\underline{201})_4 \Rightarrow 33, \text{fulfilled } (32 < 33).
\end{aligned}$$

The modified VLL is $L_0 = 31$:

$$\text{Level } 1: \quad (31)_{10} = (\underline{1}33)_4 \Rightarrow 1, (33)_{10} = (\underline{2}01)_4 \Rightarrow 2, \text{fulfilled } (1 < 2)$$

With the initial VLL, the branch algorithm recognizes the branch at the 3rd level, whereas the modified VLL detects a branch at the 1st level, which is correct.

Now a few optimisation issues are explained. For some reason, the todo-list is sorted each step in the old version of this routine. The number of these candidates for any level is $8^{\texttt{level}}$. This can be very expensive, when the branch algorithm enters high levels. The same effect can be obtained when the 1st level branch candidates are sorted. If this candidates are not recognized as a branch node, then they are split into their 2nd level children and are put in the todo-list. This list is sorted automatically, because the children are calculated in sequence. In Example 5.2.3, the reason for the reduction is portrayed.

Example 5.2.3. The sorted 1st level branch candidates can look like: $\{(0)_4, (1)_4, (2)_4, (3)_4\}$. Assuming that the algorithm does not detect any candidate as a branch node, then every key is resolved in its child cells. As a consequence, the 2nd level branch candidates are intrinsically sorted:

$$\{\underbrace{(00)_4, (01)_4, (02)_4, (03)_4}_{(0)_4}, \underbrace{(10)_4, (11)_4, (12)_4, (13)_4}_{(1)_4}, \ldots$$
$$\ldots, \underbrace{(20)_4, (21)_4, (22)_4, (23)_4}_{(2)_4}, \underbrace{(30)_4, (31)_4, (32)_4, (33)_4}_{(3)_4}\}.$$

This process can be continued and everytime a sorted list is the result. The essential condition is, that the initial list of the 1st level candidates is sorted.

Another part that can be tuned, is the determination of all first level keys, which are required as a initial condition for the branch algorithm. So far, for all twigs and leafs the level is computed. The overall number of nodes is approximately $8/7 \cdot N/P$ (homogeneous) and $2 \cdot N/P$ (inhomogeneous), as described in Section 3.1.2, page 25.

The new idea is to use the mapping property of the hash function. As a reminder, keys on a lower or equal level than h, are directly mapped on the address (see Section 3.1.2, page 25). Instead of an iteration over all local nodes, artificial 1st level keys are constructed. If a hash entry for the artificial key exists (`address` = `key`), then a 1st level branch candidate is detected. In Example 5.2.4, the mapping property of the hash function is shown for an exemplary 1st level key.

Example 5.2.4. We take the hash constant 31 ($h = 5$) and any 1st level key like $(2)_4$. Then following holds:

$$\textsc{And}((000010)_2, (011111)_2) \Rightarrow (000010)_2 = (2)_4.$$

Once the new concept is integrated into the branching algorithm, further optimisation topics are considered. The next novel development is a tight a-priori estimation of the number of local and global branch nodes for the VLD concept.

5.3 The Cross Sum Branch Node Estimator

With the original branch concept a tight a-priori estimation of the number of branch nodes is still impossible. The branch nodes at the edges of the local domain are grained. That cannot be accurately forecast. The concept of VLD allows a tight a-priori estimation, which is called the CSBE (Cross Sum Branch Node Estimator). This novel estimation is introduced in the following.

The VLD of the task p is limited by the VLL L_{p-1} and L_p. These limits are calculated from keys of the level `nlev`. Hence both VLL are on the highest refinement level, too. Overall the VLD is assembled from $n_{cells} := L_p - L_{p-1} + 1$ `nlev`-cells, which should be apportioned within the set of branches. The idea for the estimation comes from the octal-representation of n_{cells}:

$$n_{cells} = \sum_{i=0}^{\texttt{nlev}} \gamma_i 8^i = (\gamma_{\texttt{nlev}} \cdots \gamma_0)_8 \,.$$

Metaphorically, a number is made up of portions with the size 8^i and the multiplicity γ_i. If all portions are added, then the composite number results. Also branches can be interpreted as portions, that include a certain number of `nlev`-cells (see Definition 3.1.2, page 28(ii)). Analogously, γ_i describes the number of branches for a given size 8^i. But it has to be paid attention to the level of this branch. A branch at the 1st level includes $8^{\texttt{nlev}}$ cells from the level `nlev`. A 2nd level branch includes $8^{\texttt{nlev}-1}$ thereof. Finally, a branch at the level `nlev` includes exactly $8^0 = 1$ `nlev`-cell. According to this, the highest digit $\gamma_{\texttt{nlev}}$ is the multiplicity of large 1st level branches and γ_0 of branches at `nlev`. Consequentially, $\gamma_{\texttt{nlev}-\texttt{level}}$ describes the number of branches at a given level. The cross sum brings an estimation for the local number of branches $\tilde{B}_{local}$ and gives the estimator the name:

$$\tilde{B}_{local} = \sum_{i=0}^{\texttt{nlev}} \gamma_i.$$

This is an estimation, because in practice only branches are counted, that include at least one particle, which is not guaranteed here. But with an increasing number of particles and only large branches, this estimation should be very tight. This is because the probabilty is lower, that a branch does not contain a particle. A closer investigation of this topic can be found in Section 5.6.

The problem of this (simple) estimation is, that the branches are starting and ending at discrete positions, which do not match the level. The condition (iii) of the Definition 3.1.2, page 28 is bruised. Geometrically, it means that abstract instead of cubic forms are embraced, which may ultimately falsify the number. To outsource this error, the branches must be aligned to a reference point. Even the largest branches must be aligned. Hence a respective reference point must be found. Finally, the largest limit R_p, which is located within the VLD and applies to the lowest branch level, is needed. Following well-known formula, now the VLL are used instead, arises

$$\tau_{R_p} = \max\left\{ i \ \in \ \{0, \ldots, \texttt{nlev}\} : L_{p-1} < c \cdot 8^i \leq L_p \right\},$$

which can be solved by Equation 5.4. Let τ_{R_p} be the level of the reference point that equals the optimal branch level, then

$$R_p \in \left\{ \left\lfloor \frac{(L_{p-1}+1)}{8^{\tau_{R_p}}} \right\rfloor 8^{\tau_{R_p}}, \ldots, \left\lfloor \frac{(L_p)}{8^{\tau_{R_p}}} \right\rfloor 8^{\tau_{R_p}} \right\}.$$

Now we can represent the VLD as

$$D_1 := R_p - L_{p-1} = \sum_{j=0}^{\texttt{nlev}} \alpha_j \cdot 8^j, \quad D_2 := L_p - R_p = \sum_{j=0}^{\texttt{nlev}} \beta_j \cdot 8^j + 1 \ ,$$

and all conditions of the branch definition are fulfilled. The overlap condition is guaranteed by VLD. The digits α_j and β_j describe the number of branches in the respective sub domain for every level. With it, the correct number of local branches can be estimated

$$\tilde{B}_{local} = \sum_{i=0}^{\texttt{nlev}} (\alpha_i + \beta_i).$$

If we use a reduction operation across all MPI ranks, we can establish a tight upper bound $\tilde{B}$ of the global number of branches, using

$$\tilde{B} = \sum_{p=0}^{P-1} \sum_{j=0}^{\texttt{nlev}} (\alpha_j^{(p)} + \beta_j^{(p)}).$$

The next example shows, that a reference point is essential for a correct estimation.

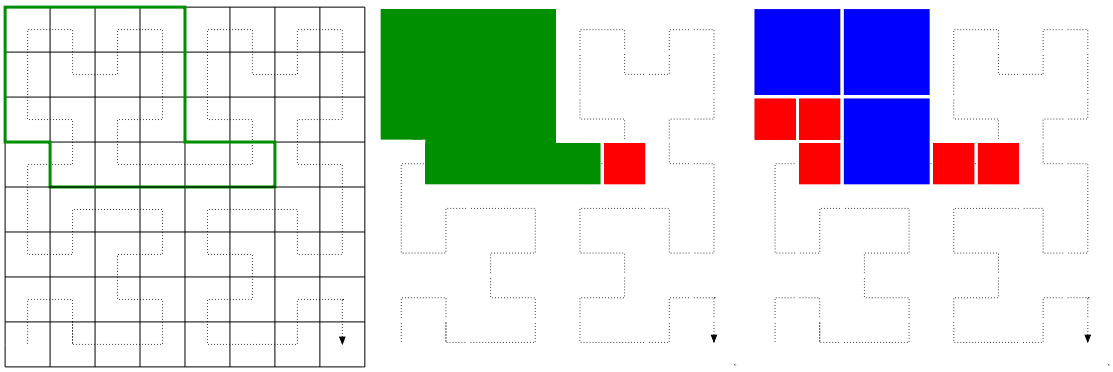
Example 5.3.1. We take the VLD $\{17, \ldots, 33\}$. The simple estimation provides

$$(33 - 17 + 1)_{10} = (17)_{10} = (010001)_2 = (101)_4 \Rightarrow 1 + 0 + 1 = 2 \text{ branches},$$

while the local CSBE provides 8 branch nodes:

$$D_1 = (32 - 17)_{10} = (15)_{10} = (001111)_2 = (033)_4 \Rightarrow 0 + 3 + 3 = 6 \text{ branches}$$
$$D_2 = (33 - 32 + 1)_{10} = (2)_{10} = (000010)_2 = (002)_4 \Rightarrow 0 + 0 + 2 = 2 \text{ branches}.$$

The problem necessity of a reference point, for the alignment of the branch nodes is geometrically shown:



(a) VLD (b) simple estimation (c) CSBE

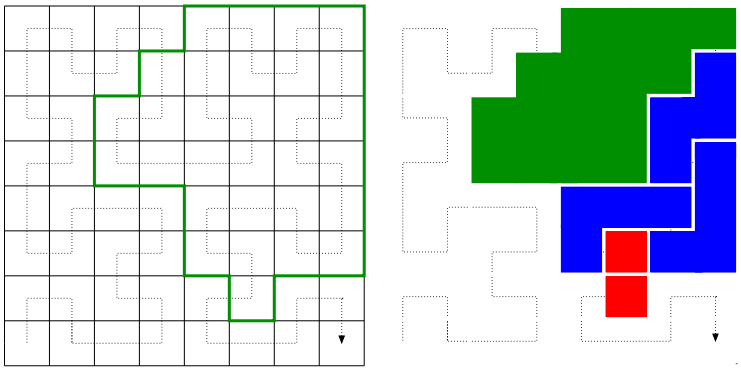
Figure 5.7: Necessity of a reference point as an essential ingredient for the Cross Sum Branch Node Estimator. A 1st level branch is green, a 2nd level branch blue and a 3rd level branch red colored. The local CSBE provides correct branch nodes.

Now the CSBE is applied to the standard example, which was pursued through the entire chapter (see Example 5.1.1). Once again, this example demonstrates the usage of reference points. In some situations several reference points are available.

Example 5.3.2. The VLD is $\{27, \ldots, 56\}$. The simple estimation provides:

$$(56 - 27 + 1)_{10} = (30)_{10} = (011110)_2 = (132)_4 \Rightarrow 1 + 3 + 2 = 6 \text{ branches}. \tag{5.5}$$

Coincidentially, the number of branches for the simple estimation equals the CSBE (see Example 5.3.1). The abstract shape of the branch nodes is illustrated in Figure 5.8.



(a) VLD (b) "branch nodes"

Figure 5.8: Abstract shaped branch nodes for the simple estimation and the standard example.

In this VLD two reference points $\{R_p\} \in \{32, 48\}$ are located. For $R_p = 32$: $D_1 = D_{1,1} \cup D_{1,2} = \{27, \ldots, 31\} \cup \{32, \ldots, 56\}$, the CSBE provides:

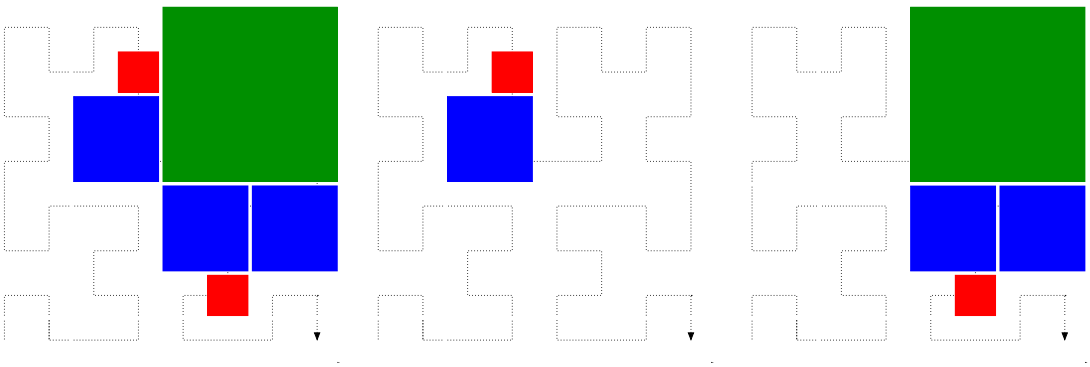
$$D_{1,1} = (32 - 27)_{10} = (5)_{10} = (000101)_2 = (11)_4 \Rightarrow 0 + 1 + 1 = 2 \text{ branches}$$
$$D_{1,2} = (56 - 32 + 1)_{10} = (25)_{10} = (011001)_2 = (121)_4 \Rightarrow 1 + 2 + 1 = 4 \text{ branches.}$$

Overall, $2 + 4 = 6$ local branch nodes are estimated. The branch nodes for every sub domain are displayed in Figure 5.9. For $R_p = 48$ the decomposition is: $D_2 = D_{2,1} \cup D_{2,2} = \{27, \ldots, 47\} \cup \{48, \ldots, 56\}$. The branches for this decomposition are shown in Figure 5.10. In turn the CSBE results in 6 branch nodes:

$$D_{2,1} = (48 - 27)_{10} = (21)_{10} = (010101)_2 = (111)_4 \Rightarrow 1 + 1 + 1 = 3 \text{ branches}$$
$$D_{2,2} = (56 - 48 + 1)_{10} = (9)_{10} = (001001)_2 = (021)_4 \Rightarrow 0 + 2 + 1 = 3 \text{ branches.}$$

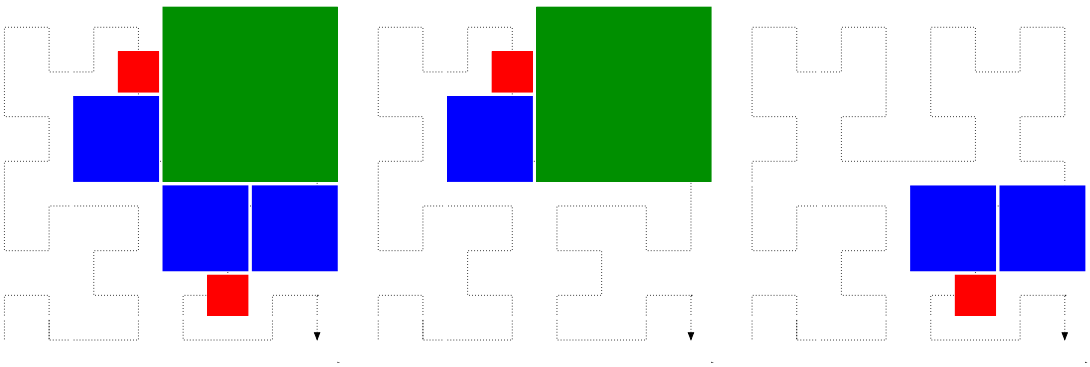
A novel branch estimation was introduced in detail. Without any knowledge of the tree, the branch nodes can be constructed a-priori. This leads to the conclusion, that a new branch algorithm can be built on the basis of CSBE. An estimation is also derived for the original branch concept. The same terms and conditions, like the reference point, are needed. We get the estimation by the substitution of the VLLs and the real limits, $k_{p,min}$ and $k_{p,max}$, in various equations. The following equation holds:

$$D_1 := R_p - k_{p,min} = \sum_{j=0}^{\texttt{nlev}} \alpha_j \cdot 8^j, \quad D_2 := k_{p,max} - R_p = \sum_{j=0}^{\texttt{nlev}} \beta_j \cdot 8^j + 1 \ .$$



(a) D_1 (b) $D_{1,1}$ (c) $D_{1,2}$

Figure 5.9: Estimated branch nodes with the reference point $R_p = 32$ and the respective sub domains.



(a) D_2 (b) $D_{2,1}$ (c) $D_{2,2}$

Figure 5.10: Estimated branch nodes with the reference $R_p = 48$ and the respective sub domains.

5.4 Introduction of a Novel Branching Algorithm

The CSBE provides the number of local branches for each level. Based on this information, the position, equivalent to the key of every branch node, can be derived. One further fundamental insight is, that as the branch size decreases, the level increases monotonic, respectively, from the reference point R_p. Since this point was calculated by the VLL, which are keys of the level `nlev`, the reference point is also at this level. Furthermore, the entire local domain was split in D_1 and D_2. For both sub domains the branches are calculated separately.

The idea is to calculate any `nlev`-key lying inside the branch domain (see Definition 3.1.2, page 28) and calculate the appropriate parent cell applying the level of the branch (see Example 5.2.1).

Without loss of generality this is described for D_1, which is located prior (left) to the reference point. A pointer r is set to the reference point. Then the number of `nlev` cells, which are contained in the branch, is subtracted from this pointer. For this value, the appropriate parent cell (the level is known) is calculated and the key of the branch results. Iteratively, for every branch node at any level this simple procedure is processed. In every case the pointer is the first `nlev` cell of the current branch node. For D_2 the number of `nlev` cells is added to the counter. The novel branching algorithm can be directly performed after the `domain decomposition`. Finally, the following equation holds for a correct pointer r and a branch at the appropriate level:

$$\texttt{branchKey} = \textsc{And}((8^{\texttt{nlev+1}} - 1) - (8^{\texttt{level+1}} - 1), r - 8^{\texttt{nlev}-\texttt{level}}). \tag{5.6}$$

Since the CSBE is only an estimation, branch nodes are introduced, that are not present in the local tree. Hence, the hash address is calculated for the branch key (see Equation 3.1, page 25). If the key exists in the hash table (the address is calculated), then a branch node is identified. Naturally, this can be done even after the `local tree` construction. The additional amount of time for the construction of branch nodes, which are not present in the local tree vanishes, because the tightness of CSBE (see Section 5.6.3).

The algorithmic complexity is of the order $\mathcal{O}(\texttt{nlev})$, because branch nodes at all levels must be resolved, since one missing digit in the representation of D_1 or D_2, $\alpha_i = 0$ or $\beta_i = 0$ does not induce that the local tree does not contain a branch node at a deeper level.

Because the run time for this step can be neglected, a comparison with the old algorithm is not performed here. The novel branching algorithm is faster then the original code with the old concept and the original code with VLD. Furthermore, it does not need an initial state and the construction of child keys. The local tree does not have to be traversed.

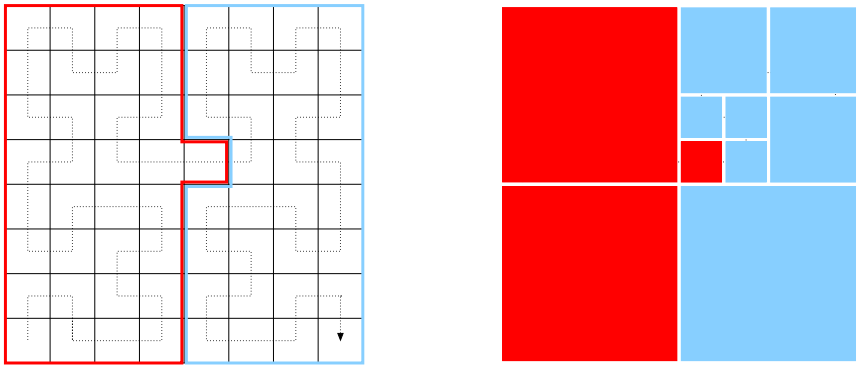
5.5 Capabilities of Virtual Local Domains

Besides the advantages of the VLD concept, one must also know its limits. The success of this novel branching concept heavily depends on the particle dispersal in the `domain decomposition`. But this issue holds for the original concept as well.

In Figure 5.11 a worstcase scenario, regarding the distribution of the particles after the `domain decomposition`, is presented. A single key, at the maximal refinement level (here `nlev=3`) crosses the highest possible VLL and extends into a 1st level branch node. For a one sided infiltration of a branch node at level l_1 by another branch node at level l_2 the number of additional branches is

$$\left(2^3 - 1\right) |l_1 - l_2| \equiv (8 - 1) |l_1 - l_2| . \tag{5.7}$$

For this reason, a large number of branch nodes (in this 2D case $(2^2-1)\,|1-3| = 6$ extra branches result) are introduced. If the single key would be shifted to the adjacent task (task 0), then 4 instead of 10 global branch nodes would result. But the load-balancing could be affected in a negative way by this shift.



(a) VLD (b) branch nodes

Figure 5.11: Capability of the VLD concept. A worst case scenario is presented.

For large scales, where the branch `exchange` is a major part of the total run time, a mixed load-balancing strategy can be imagined. Currently, the number of interactions of the previous time step is decisive for the weights of the sort in the `domain decompositon` (see Section 3.1.1, page 23). But for large scales, the `exchange` instead of the `tree traversal` dominates the total run time. This `exchange` benefits from the shift of some particles to adjacent tasks, because the global number of branch nodes decreases. According to this, it is decided whether the particle is shifted or not.

Another issue that affects VLD, is the selection of the task count. At least for the homogeneous case, the concept of VLD benefits from the selection of 8^x tasks, because the VLL are automatically aligned to the maximum branch level.

Nevertheless, the VLD concept, regarding the number of branch nodes, is optimal. The proof is omitted, since this concept is constructed to provide the minimal number of branches. An idea for the proof is: assume that the optimal VLL is on a higher level. Then the CSBE shows that the number of branch nodes is increased, which leads to a contradiction. The selection of a VLL at a lower level produces a contradiction to the branch definition, since each branch must overlap all child cells.

5.6 Effects of Virtual Local Domains in PEPC

The effects of the VLD concept for the parallel tree code PEPC are statistically examined in this section. A comparison to the original concept is made. Especially, the local and global branch amount is intensively researched. In addition, the power of the CSBE is analysed and novel possibilities of application of the VLD concept are introduced.

5.6.1 Number of Global Branch Nodes

In this section, the global number of branch nodes is studied for VLD and the original concept. As described in Section 3.2, page 34, the global branch nodes must be stored in the local hash table of each tasks and consume a lot of memory. In addition, theses nodes are exchanged between all tasks, which prevents the scaling of a parallel HOT-scheme and the anticipated code PEPC, since their amount immensely increases for large scales. Therefore, it is a distinct topic to reduce the number of branch nodes to a minimum. In Figure 5.12 the global amount is only visualised for the inhomogeneous setup, because the difference to the homogeneous one is marginal.

N	P	**original**	**VLD**	**ratio**
$0.125 \cdot 10^6$	2	156	8	5%
$8.000 \cdot 10^6$	4096	140559	55313	39%
$2048 \cdot 10^6$	16384	999960	208538	21%

Table 5.1: Comparison of the original branch concept and the one of VLD for selected particle and task counts and an inhomogeneous particle setup. The new concept only brings a small fraction of the original branch number.

In contrast to the original concept, the number of global branch nodes does not depend on the number of local particles. Furthermore, in every case the concept of VLD produces less branch nodes. In Table 5.1, the strength of the VLD concept is underlined by the data of selected particle and task counts. According to Figure 5.12, the original concept seems to converge towards the number of global branch nodes of the VLD concept for an increasing number of tasks, but very slowly. This can be explained as follows: for an increasing number of tasks, the local particle number N/P decreases. Hence, the number of branches in the grained region at the edges of the local domain is reduced. For a

vanishing number of special cases and an extraordinary small number of local particles both concepts are equal. However, this does not appear in practice. The number of global branches for the original concept depends on the number of local particles N/P and the number of tasks P. In contrast, the order of the global number of branch nodes for the VLD concept is $\mathcal{O}(P)$. This yields an $\mathcal{O}(1)$ behaviour of the number of local branch nodes and is further investigated in the next section.

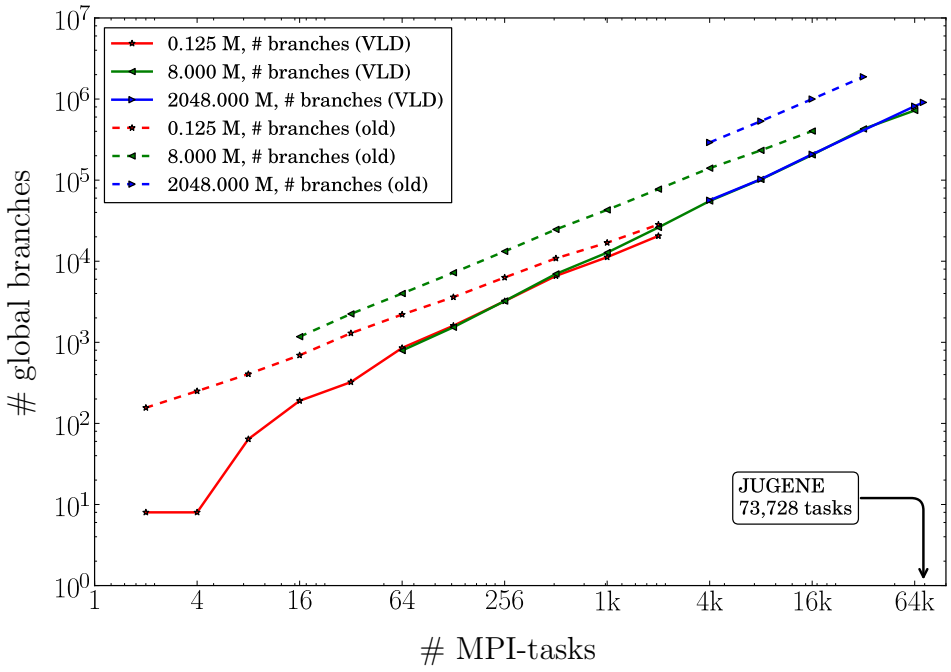


Figure 5.12: Number of global branch nodes for the original and the VLD concept for an inhomogeneous setup. The novel concept provides a magnitudes smaller number of global branches and does not depend on the local particle number N/P.

For $P = 16384$, approximately 800000 global branches are saved. Theoretically, without the provision for other factors, in each local hash table 800000 entries are saved. This can be used for additional particles. Assume that PEPC is at the memory limit, then approximately $13 \cdot 10^9$ additional particles could be simulated. Moreover, the benefit for the `exchange` is studied. In Figure 5.13 the timings are shown for the inhomogeneous case. Naturally, the times for the VLD concept are smaller than for the original concept, since less data (global branch nodes) is interchanged. For the original concept, the separate curves for different local particle counts are displaced (dependy to the local particle number). The `exchange` timings satisfy the curves for the global branch nodes, and are displaced as well. This fact underlines the MPI_ALLGATHERV dependency on the data volume.

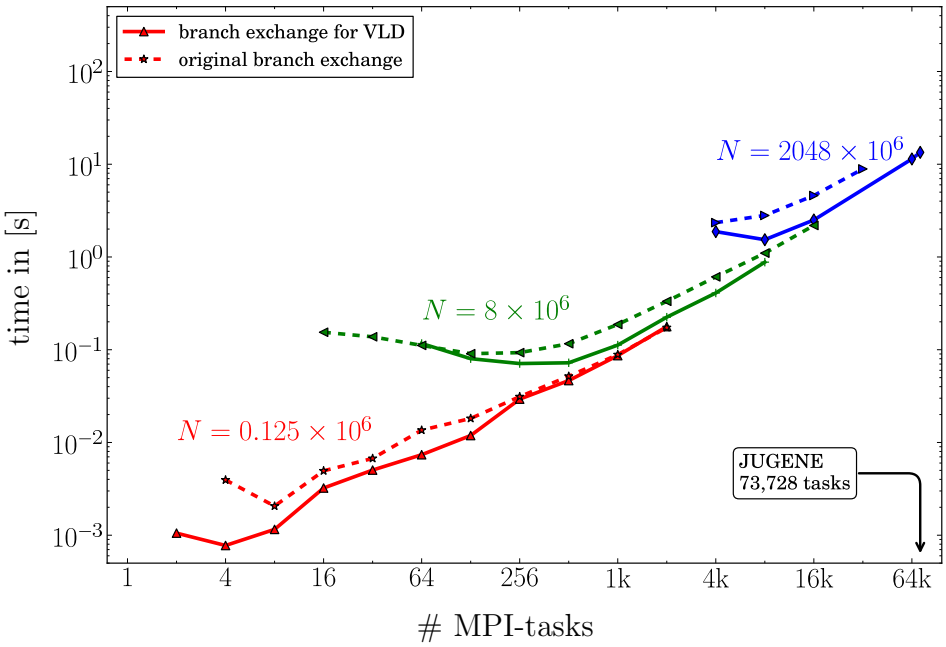


Figure 5.13: Comparison of the orginal and VLD concept. The time consumption of the `exchange` step is displayed. The time for the `exchange` step is reduced by the novel VLD concept.

For the novel concept, the `exchange` curves are placed on an imagined line, because the local particle number does not influence the number of global branch nodes as seen in Figure 5.12. In Table 5.2, the time for the `exchange` step is shown for selected task and particle counts.

N	P	**original** [s]	**VLD** [s]	**ratio**
$0.125 \cdot 10^6$	4	$0.39 \cdot 10^{-2}$	$0.77 \cdot 10^{-3}$	20%
$8.000 \cdot 10^6$	4096	0.61	0.41	67%
$2048 \cdot 10^6$	16384	4.60	2.51	55%

Table 5.2: Comparison of the orginal concept and VLD for the time consumption of the `exchange` step for selected task and particle counts. This time is reduced by the novel VLD concept.

As a consequence, the overall scaling of the code benefits from the reduction of the time of the `exchange` step. Furthermore, the memory reduction, which is caused by the VLD approach allows larger scales. This is because many global branches are saved and do not have to be integrated into every local hash table. In Figure 5.14, the scaling for PEPC is shown for maximal 73728 MPI-tasks. Linked with the hybrid ansatz, with 4 threads per MPI-task, the entire system JUGENE is utilized. For the future, the new branch concept

with additional modifications (presented in Section 5.6.4) promises extreme scales up to 294912 MPI-tasks.

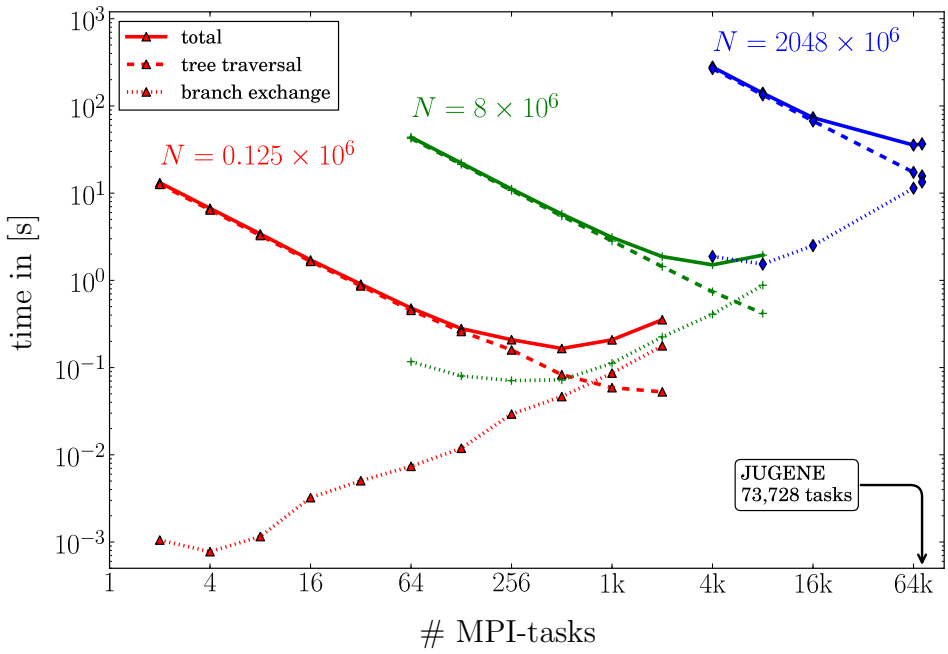


Figure 5.14: Scaling of the code with the Virtual Local Domain concept for an inhomogeneous setup up to 73728 MPI-tasks for selected particle setups and relevant steps of the HOT-sheme. The dotted lines marks the `exchange`. The dashed lines show the `traversal`. Solid lines correspond to the total run time. The novel VLD approach provides less global branch nodes. Therefore, the code can reach larger scales (regarding memory). The scalibity benefits from a reduced `exchange` time.

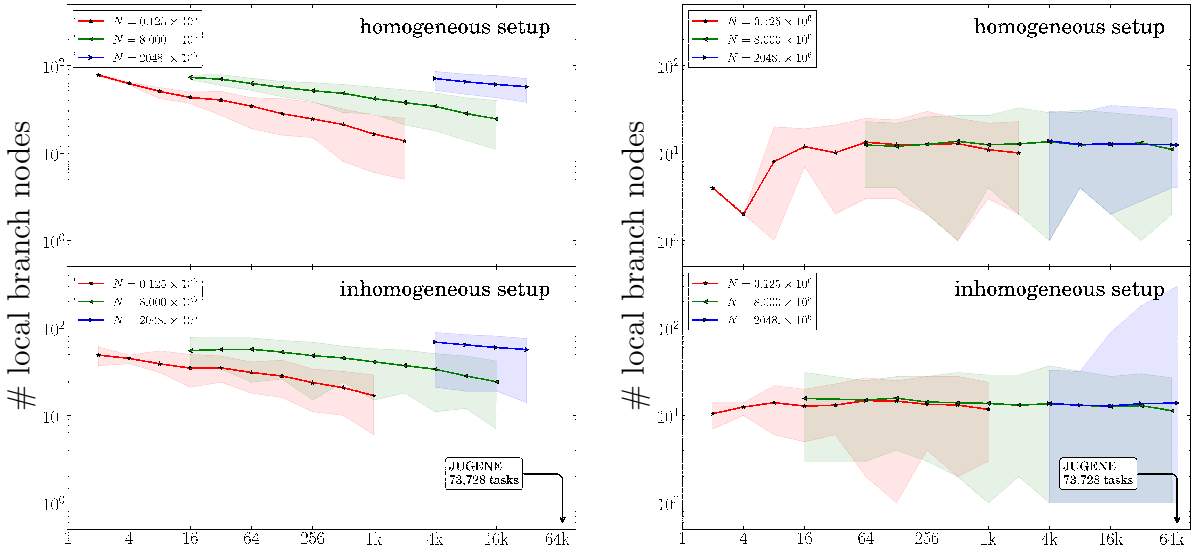
Moreover, other steps of the HOT-scheme are influenced by the novel VLD concept:

- The construction of global fill nodes benefits from the novel concept, since the branch levels are smaller and less nodes are introduced.
- The calculation of the pseudo-particle properties does not benefit directly, because the number of nodes is the same for both concepts. However, the pseudo-particle information of the branch nodes is also broadcasted, but this is respected in the `exchange` step.
- For the `tree traversal` the new concept provides more communication. For the original concept with its large number of small branches, the request is made automatically. In contrast, the large branches of the VLD concept must resolved into its children and requested from the owner. As shown in Chapter 4 the communication was reduced by the Hilbert-curve. This has no significant influence on the run time

of the `tree traversal`. Therefore, the concept of VLD does not bring any disadvantages for the run time of the `tree traversal`. This can be concluded by the comparison of Figure 5.14 and Figure 3.14, page 36.

5.6.2 Local Branch Nodes

This sections deals with the effect of the VLD concept on the number of local branch nodes. In Figure 5.15, the average number, with the respective minimum and maximum region is shown for both setups.



(a) original (b) VLD

Figure 5.15: Average number of local branch nodes and the respective minimum and maximum regions for the original concept and VLD for a homogeneous and inhomogeneous setup. For the VLD concept for both setups the average local branch number remains constant for an increasing task count. The local branch amount of the original concept are magnitudes larger and decreases for increasing scales.

For VLD, the local amount of branch nodes remains constant for both setups and an increasing number of tasks for every particle number N. Especially, the power of 8 aligned minima, as described in Section 5.5, can be seen for the homogeneous setup. For various task counts, the minimal local branch number, which is given by the definition, is reached. For the inhomogeneous setup, the necessity of a tight branch estimation is shown by large local maxima for large scales. This results from VLLs lying within dense regions, where the tree is very deep. Despite of this, VLD compensates the deviation between the separate

tasks and provides a constant number of local branch nodes for every setup. Thus, the number of local branches is $\mathcal{O}(1)$, as concluded in the previous section. For every local domain, approximately 10 branch nodes are needed. Since this number is constant, the branch amount can be estimated by $10P$, without the evaluation of the CSBE.

In fact, the average local amount of branches decreases for the original concept, but it is magnitudes larger than for the novel VLD concept. This issue can be explained by the decreasing number of local paticles N/P: the probabilty, that a branch is located within the grained region at the edges of the local domain reduces. For the original concept, no significant difference can be seen between both setups. A small deviation of this concept signifies, that the local branches heavily depend on the grained region at the edges of the local domain. Whereas the large deviation of the novel concept shows an adjustment to the optimal VLD.

5.6.3 Power of the Cross Sum Branch Node Estimator

The advantages of the VLD concept in comparison to the original concept were shown and legitimated in the previous sections. Now the power of the CSBE is researched. Additionally, the adapted CSBE for the original concept is studied. In Figure 5.16, the CSBE for the homogeneous setup, and in Figure 5.17, the CSBE for the inhomogeneous one, is shown.

N	P	**VLD**	**CSBE**	**coverage**
homogeneous				
$0.125 \cdot 10^6$	2	8	8	100.00%
$8.000 \cdot 10^6$	4096	55313	55412	99.82%
$2048 \cdot 10^6$	16384	208538	208538	100.00%
inhomogeneous				
$0.125 \cdot 10^6$	2	21	29	72.41%
$8.000 \cdot 10^6$	4096	55791	60667	91.96%
$2048 \cdot 10^6$	16384	210768	220262	95.67%

Table 5.3: Coverage of the CSBE for the VLD concept. For the homogeneous case the coverage is excellent. An acceptable coverage is observed for the inhomogeneous case.

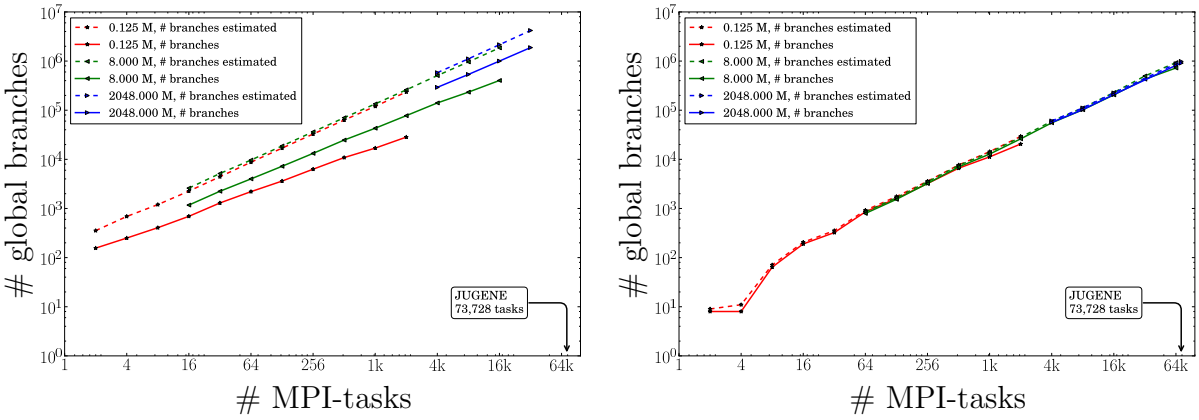
For the homogeneous case, the coverage of the CSBE is almost 100 percent and the curve in Figure 5.16 cannot be separated from the real global branch nodes. For an inhomogeneous case, the estimation is not so tight, but still arround 90 percent. As mentioned above, the VLLs can be located within dense regions with a high tree depth. Hence the CSBE cannot be expected to be so tight, because branch nodes are estimated, that do not contain a particle. For an increased local particle number N/P, the CSBE increases, because the probability decreases, that an estimated branch does not contain a particle. This can be seen in Table 5.3. Overall, this observation leads to the conclusion,

that the number of global branch nodes can be estimated very tightly. Furthermore, one challenging part for the estimation of the memory amount for the hash table was clarified (see Section 3.2, page 34).

N	P	**original**	**CSBE**	**coverage**
homogeneous				
$N = 0.125 \cdot 10^6$	2	156	354	44.07%
$N = 8.000 \cdot 10^6$	4096	140559	506645	27.74%
$N = 2048 \cdot 10^6$	16384	999960	2152944	46.45%
inhomogeneous				
$0.125 \cdot 10^6$	2	98	297	33.00%
$8.000 \cdot 10^6$	4096	138472	455257	30.41%
$2048 \cdot 10^6$	16384	982068	1937427	50.69%

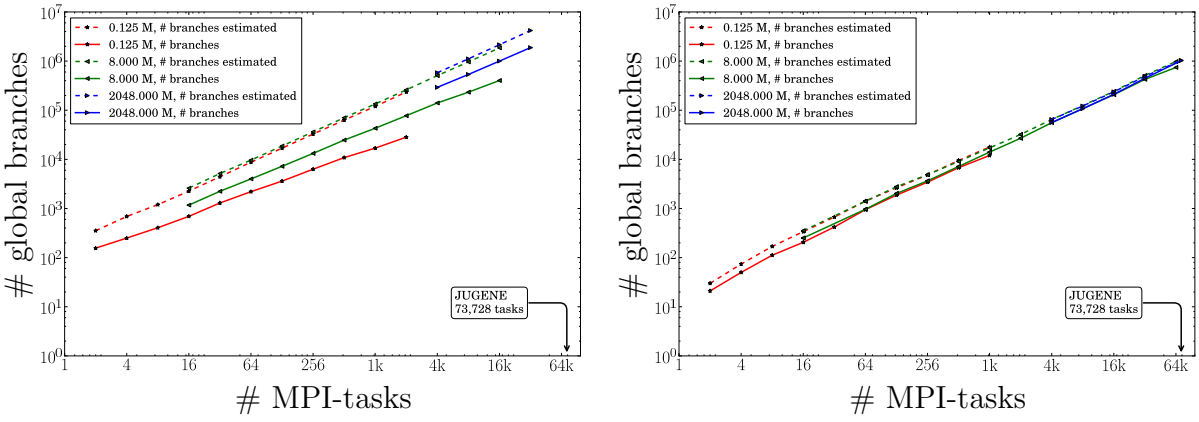
Table 5.4: Coverage of the adapted CSBE for the original concept. The coverage is low. Hence in practice, many memory would be wasted. For this reason, the adapted CSBE is does not yield an accurate prediction of the branch nodes and the memory consumption of global branches in every local hash table.

In every case, the adapted CSBE for the original concept purveys a deficient estimation. The grained branch nodes at the edges of the local domain results in a large number of estimated branch nodes, which are not present in the local tree. E.g. for the case with $P = 16384$, an absolute error of approximately 1000000 branch nodes results for every setup. In Table 5.4 the estimation is listed for selected task and particle counts. In a nutshell, the adapted CSBE equals for both setups and the estimation is useless in practice, since the coverage is low and a large abolute error is observed. The concept of VLD in conjuntion with the CSBE provides a tight a-priori estimator for the local and global branch nodes. The branches for the homogeneous case are estimated more tight.



(a) original concept with adapted CSBE (b) VLD and CSBE

Figure 5.16: Cross Sum Branch Node Estimator for the homogeneous setup and both branching concepts. For the VLD concept, the CSBE is very tight. The adapted CSBE provides a bad estimation.



(a) original concept with adapted CSBE (b) VLD and CSBE

Figure 5.17: Cross Sum Branch Node Estimator for the inhomogeneous setup and both branching concepts. Likewise the adapted CSBE does not estimate tight. Contrary to the estimation for the original concept, the CSBE in conjunction with the VLD concept has a high coverage. For the homogeneous case, the coverage (the power of the CSBE) is larger.

5.6.4 Perspective of Virtual Local Domains

The novel concept of VLD provides a lot of further advantages and possibilities of application. In this section, two possible improvements for the HOT-scheme are introduced.

The first idea depicts a further reduction of the global branches for the individual task. It is possible to calculate all global branch nodes on each task for all other tasks, by the knowledge of all limits, which were setup in the `domain decomposition`. Since the average number of local branch nodes is constant, this requires an effort of $\mathcal{O}(P)$. Regarding memory consumption, the calculation of all global branch nodes for each task is not a problem, because the CSBE provides the global amount of branch nodes, and the array size is known. The CSBE and the anticipated branch finding algorithm are implemented very fast. Accordingly, the time consumption for this step should be small. After all branch nodes of the other tasks are known, it could be geometrically estimated, as shown in Figure 5.18, which non-local branch nodes are required for the `force summation`, the `tree traversal` respectively, for each task. The geometric criterion is a modification of the Barnes-Hut MAC (see Equation 2.3, page 9). The geometrical MAC fits to a worst case. The distance d for the BH MAC is determined by the particle position and the center of charge of the pseudo-particle. Hence, for the novel geometric estimation, which valuates two pseudo-particles, the worst case for both center of charges must be taken. This is the minimum distance between the cells. Besides the reduction of the memory amount for the global branch nodes in the local hash table, all interaction partners could be obtained a-priori. Hence, the set of interaction partners could contain some not required nodes, because the CSBE is an estimation.

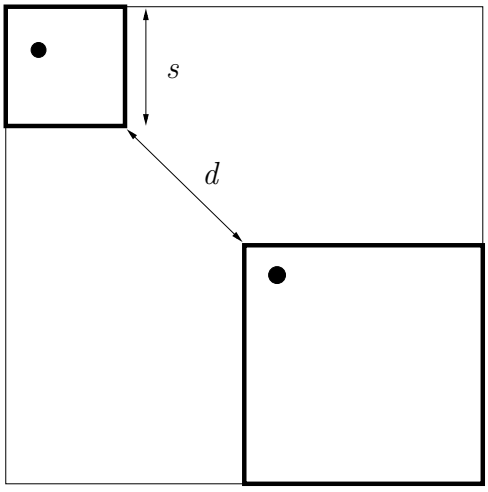


Figure 5.18: Geometric estimation for the purpose of a further reduction of global branches in the local hash table based on the VLD concept. The black circles mark the real center of charges of the pseudo-particle in the specific cell. The distance d is a worst case estimation.

Based on the knowlegde of all branch nodes, which are required for the `tree walk`, the number of requested nodes could be estimated. By this, all parts of the memory estimation for the local hash table (see Section 3.2, page 34) are known. In the current implementation this is done by an artificial input parameter, which usually allocates the available memory. Hence, for small local particle numbers, unneccessary memory allocation, which costs a lot of time, can be saved. However, the number of hash collisions could be affected, which then has to be studied in another context.

The second idea could optimise the calculation of global fill nodes above the branch level. The construction of local fill nodes intrinsically happens when the the `local tree` is built. The current implementation disregards these nodes, because they may be incomplete. A novel idea would be to retain all local fill nodes. Now, the local hash entries, which describe the fill nodes could be merged:

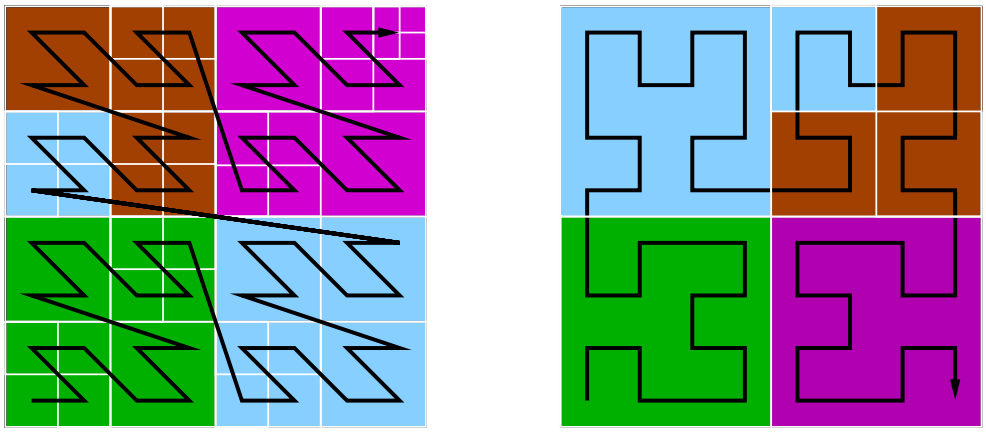
- The `leaves` are added.
- The childcode is linked by an OR operation.

A problem is the determination of the appropriate range in the hash table. Here, the novel concept of VLD with the CSBE could be a solution. Since the number of global fill nodes is minimised for the VLD concept, this novel idea could be efficiently applied in practice.

Chapter 6

Compendium

In this chapter, the massive impacts of this master thesis for the parallel BH tree code PEPC are summarized. Being free of any change for the front-ends, the kernel of this application was modified by two relevant approaches. A comparison between the original version of the PEPC kernel and the one after this thesis is shown Figure 6.1.



(a) Morton-curve and original branch concept

(b) Hilbert-curve and VLD concept

Figure 6.1: Illustration of the basic modification of the PEPC kernel, that were introduced in line with this thesis and applied to the code. Through the Hilbert-curve, the data-locality is improved and divided domains are disregarded. The novel VLD concept reduces the number of branch nodes to a remarkable optimum.

The first modification is the replacement of the Morton- by the Hilbert-curve. As shown statistically, the Hilbert-curve provides an immense reduction of communication in the irregular structure of the combined `tree traversal` and `force summation` in the hybrid MPI-PTHREADS approach. In addition, fast Hilbert-mapping algorithms were introduced and generalized on all possible 2D Hilbert-patterns as well as novel SFCs.

The modification with the highest outcome, is the novel branch concept. The VLD concept provides an optimal reduction of the branch data structure in a data-distributed parallel BH tree code. Accordingly, the memory consumption of the global branch nodes in every local hash table is minimised, which results in larger N-body simulations at extreme scales. As a consequence the code scaling is improved. Moreover, a tight a-priori estimation, the CSBE, for the local and global branch nodes is derived, achieving nearly perfect coverage for homogeneous and inhomogeneous setups. For future developements the novel VLD concept introduces a multitude of optimisation possibilities: Based on the CSBE and the VLD concept the global data amount in the local hash table can be further reduced. In combination with the Hilbert-curve, with a minimum of global communication partners, as the appropriate basis, state of the art parallel N-body simulations can reach remote targets.

Finally, the introduced VLD concept in conjunction with the hybrid traversal approach [63] prepares the code for extreme-size N-body simulations at extreme-scales of current and upcoming massively parallel supercomputing architectures.

Appendix A

Source Code of Selected Algorithms

```
! Partial keys
integer*8 :: ix, iy, iz

! Degree of quantization of the space (=21)
integer*8, parameter :: nlev=nlev

! The Morton-key on the curve Z^3_nlev
integer*8 :: zkey

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! construct Morton-derived key
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! For all refinement levels
do i=0,nlev-1
   ! Appending Hilbert-order to hkey
   zkey = ior(ishft(zkey, 3), 4_8*ibits(iz, nlev-1_8-i, 1_8) &
                            + 2_8*ibits(iy, nlev-1_8-i, 1_8) &
                            + 1_8*ibits(ix, nlev-1_8-i, 1_8) )
end do
```

Listing A.1: FORTRAN90 : Morton-mapping from 3D to 1D. Generating Morton-derived keys for a fixed degree of space quantization with the binary interleave operation.

```
! Partial keys
integer*8 :: ix, iy, iz

! Degree of quantization of the space (=21)
integer*8, parameter :: nlev=nlev

! The Morton-key on the curve Z^3_nlev
integer*8 :: zkey

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Construct partial keys
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Inverse binary interleaving operation
do i=0,nlev-1
   iz=ior(ishft(ibits(zkey, 3*i, 3), i),z)
   iy=ior(ishft(ibits(zkey, 3*i, 3), i),y)
```

```
   ix=ior(ishft(ibits(zkey, 3*i, 3), i),x)
end do

! Cut unneccessary bits
iz=ibits(z, 0, nlev)
iy=ibits(y, 0, nlev)
ix=ibits(x, 0, nlev)
```

Listing A.2: FORTRAN90 : Morton-mapping from 1D to 3D. Generating partial keys from any Morton-derived key for a fixed degree of space quantization with the inverse binary interleave operation.

```
! Define the 1st order Hilbert-curve (inverse: for O(1) access)
! Equivalent for all patterns
integer*8, parameter :: CI(0:7) = [0, 1, 3, 2, 7, 6, 4, 5]

! Define the Hilbert-gene (evolvement rule table for rotations and reflections)
! Depends on the pattern and can be hard-coded implemented, if a fixed pattern
! is selected
integer*8, parameter :: G(0:7,0:1) =
   reshape([5, 6, 0, 5, 5, 0, 6, 5, 0, 0, 0, 5, 0, 0, 6, 5],shape(G))

! Partial keys
integer*8 :: ix, iy, iz

! Degree of quantization of the space (=21)
integer*8, parameter :: nlev=nlev

! The Hilbert-key on the curve H^3_nlev
integer*8 :: hkey

! Octant of the Hilbert-curve derived by CI
! The Morton-order bit tripel is plugged in as index
integer*8 :: horder

! Temporal variables to save partial keys
! One variable for change of directions
integer*8 :: xtemp, ytemp, ztemp, change

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Construct Hilbert-derived key
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Copy, because construction alters original values
xtemp = ix
ytemp = iy
ztemp = iz

! For all refinement levels despite of the highest, construct the octant
! Attach it to the Hilbert-key
do i=0,nlev-2

   ! Get Hilbert-order
   ! Gathering upper bits as in Morton-mapping and combine to binary number
   ! Derive the octant of the Hilbert-curve from this Morton-tripel
   horder = CI( 4_8*ibits(ztemp, nlev-1_8-i, 1_8) &
              + 2_8*ibits(ytemp, nlev-1_8-i, 1_8) &
              + 1_8*ibits(xtemp, nlev-1_8-i, 1_8))

   ! Appending Hilbert-order to hkey
   hkey = ior(ishft(hkey, 3), horder)

```

```
    !!!!!!!!!!!!!!!!!!!!!!!
    ! Transform coordinates
    !!!!!!!!!!!!!!!!!!!!!!!

    ! Rotation: exchange coodinates
    select case (G(horder,0))
       case (5) ! = 101 = CI(zyx) --> change z and x
          change = ztemp
          ztemp  = xtemp
          xtemp  = change
       case (6) ! = 110 = CI(zyx) --> change z and y
          change = ztemp
          ztemp  = ytemp
          ytemp  = change
    end select

    ! Reflections: reverse coordinates
    ! Thus Fortran 90 does not include unsigned datatypes the highest bit must set to 0,
    ! To ensure positive partial keys
    select case (G(horder,1))
       case (5) ! = 101 = C(zyx) --> reverse z and x
          ztemp = not(ztemp)
          ztemp = ibclr(ztemp,63)

          xtemp = not(xtemp)
          xtemp = ibclr(xtemp,63)
       case (6) ! = 110 = C(zyx) --> reverse z and y
          ztemp = not(ztemp)
          ztemp = ibclr(ztemp,63)

          ytemp = not(ytemp)
          ytemp = ibclr(ytemp,63)
    end select
end do

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! For the highest refinement-level no coordinate transforms must be performed
! Because a higher level must not mapped on the curve
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Get Hilbert-order
! Gathering upper bits as in Morton-mapping and combine to binary number
! Derive the octant of the Hilbert-curve out of this Morton tripel
horder = CI(   4_8*ibits(ztemp,0_8,1_8) &
             + 2_8*ibits(ytemp,0_8,1_8) &
             + 1_8*ibits(xtemp,0_8,1_8))

! Appending Hilbert-order to hkey for hihest level
hkey = ior(ishft(hkey, 3), horder)
```

Listing A.3: FORTRAN90 : Hilbert-mapping from 3D to 1D. Generating Hilbert-derived keys for a fixed degree of space quantization with the fast 3-dimensional Hilbert-mapping algorithm.

```
! Define the 1st order Hilbert-curve
! Equivalent for all patterns
integer*8, parameter :: C(0:7) = [0, 1, 3, 2, 6, 7, 5, 4]

! Define the Hilbert-gene (evolvement rule table for rotations and mirrorings)
! Depends on the pattern and can be hard-coded implemented if the pattern is fixed
integer*8, parameter :: G(0:7,0:1) =
   reshape([5, 6, 0, 5, 5, 0, 6, 5, 0, 0, 0, 5, 0, 0, 6, 5],shape(G))

```

```
10 ! Partial keys
11 integer*8 :: ix, iy, iz
12
13 ! Degree of quantization of the space (=21)
14 integer*8, parameter :: nlev=nlev
15
16 ! The Hilbert-key on the curve H^3_nlev
17 integer*8 :: hkey
18
19 ! Octant of the Hilbert-curve derived by C
20 ! The Hilbert-order bit tripel is plugged in as index
21 integer*8 :: horder
22
23 ! One variable for change of directions
24 integer*8 :: change
25
26 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
27 ! construct partial key
28 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
29
30 ! Gather lowest bits of Hilbert-derived key
31 ! Hilbert-octant for the highest refinement level
32 horder=ibits(hkey,0,3)
33
34 ! Put adequate bits in repective partial coordinate
35 iz=ibits(C(horder),2,1)
36 iy=ibits(C(horder),1,1)
37 ix=ibits(C(horder),0,1)
38
39 ! Resolve partial keys for all refinement levels
40 ! Despite of the highest level
41 do i=1,nlev-1
42
43    ! Hilbert-octant for the i-th level
44    horder=ibits(hkey,3*i,3)
45
46    !!!!!!!!!!!!!!!!!!!!!!
47    ! Transform coordinates
48    !!!!!!!!!!!!!!!!!!!!!!
49
50    ! Reflections: reverse coordinates
51    ! Keep track of higher bits, and restore to zero
52    select case(G(horder,1))
53       case(5) ! = 101 = C(zyx) --> reverse z and x
54          x=iand(not(ix),2**(i)-1)
55          z=iand(not(iz),2**(i)-1)
56       case(6) ! = 110 = C(zyx) --> reverse z and y
57          y=iand(not(iy),2**(i)-1)
58          z=iand(not(iz),2**(i)-1)
59    end select
60
61    ! Rotation: exchange coodinates
62    select case (G(horder,0))
63       case (5) ! = 101 = C(zyx) --> change z and x
64          change = iz
65          iz = ix
66          ix = change
67       case (6) ! = 110 = C(zyx) --> change z and y
68          change = iz
69          iz = iy
70          iy = change
71    end select
72
```

```
73
74      ! Append back-transformed ("natural") bits to partial keys
75       iz=ior(ishft(ibits(C(horder),2,1),i),iz)
76       iy=ior(ishft(ibits(C(horder),1,1),i),iy)
77       ix=ior(ishft(ibits(C(horder),0,1),i),ix)
78
79  end do
80
81  ! Cut bits greater than nlev-1
82  iz=ibits(iz,0,nlev)
83  iy=ibits(iy,0,nlev)
84  ix=ibits(ix,0,nlev)
```

Listing A.4: FORTRAN90 : Hilbert-mapping from 1D to 3D. Generating partial keys out of the Hilbert-derived key for a fixed degree of space quantization with the inverse fast 3-dimensional Hilbert-mapping algorithm.

Bibliography

[1] R. Rojas, F.L. Bauer, and K. Zuse. *Die Rechenmaschinen von Konrad Zuse*. Springer, 1998.

[2] K. Zuse, F.L. Bauer, and H. Zemanek. *Der Computer - Mein Lebenswerk*. Springer, 2007.

[3] G. E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Electronics* 38.8 (Apr. 1965).

[4] S. Pfalzner and P. Gibbon. *Many Body Tree Methods in Physics*. New York: Cambridge University Press, Sept. 2005.

[5] J. Barnes and P. Hut. "A Hierarchical O(N log N) Force-Calculation Algorithm". In: *Nature* 324.6096 (Dec. 1986), pp. 446–449.

[6] J. K. Salmon and M. S. Warren. "Skeletons From the Treecode Closet". In: *Journal of Computational Physics* 111 (1 Mar. 1994), pp. 136–155.

[7] M. Keldenich. *Optimierung des Multipol-Akzeptanz Kriteriums im Barnes-Hut Baumalgorithmus*. Diploma thesis. Fachhochschule Aachen, Standort Jülich, Jan. 2008.

[8] Jülich Supercomputing Centre (JSC). visited: 08.03.2011, 21:08. URL: http://www.fz-juelich.de/jsc.

[9] Forschungszentrum Jülich GmbH. visited: 08.03.2011, 21:06. URL: http://www.fz-juelich.de.

[10] P. Gibbon. *PEPC: Pretty Efficient Parallel Coulomb-solver*. Technical Report. FZJ-ZAM-IB-2003-05. Forschungszentrum Jülich GmbH, May 2003.

[11] PEPC can be obtained from. URL: https://trac.version.fz-juelich.de/pepc.

[12] M. S. Warren and J. K. Salmon. "A Parallel Hashed Oct-Tree N-Body Algorithm". In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. Supercomputing '93. Portland, Oregon, United States: ACM, 1993, pp. 12–21.

[13] M. S. Warren and J. K. Salmon. "A Portable Parallel Particle Program". In: *Computer Physics Communications* 87.1-2 (1995). Particle Simulation Methods, pp. 266–290.

[14] P. Gibbon, R. Speck, A. Karmakan, L. Arnold, W. Frings, B. Berberich, D. Reiter, and M. Masek. "Progress in Mesh-free Plasma Simulation With Parallel Tree Codes". In: *Transaction on Plasma Science* 38 (2010), pp. 2367–2376.

[15] J. K. Salmon, M. S. Warren, and G. S. Winckelmans. "Fast Parallel Tree Codes for Gravitational and Fluid Dynamical N-Body Problems". In: *International Journal of Supercomputer Applications* 8 (1986), pp. 129–142.

[16] G. S. Winckelmans and A. Leonard. "Contributions to Vortex Particle Methods for the Computation of Three-Dimensional Incompressible Unsteady Flows". In: *Journal of Computational Physics* 109.2 (1993), pp. 247–273.

[17] G. S. Winckelmans, J. K. Salmon, A. Leonard, and B. Jodoin. *Application of Fast Parallel and Sequential Tree Codes to Computing Three-Dimensional Flows with the Vortex Element and Boundary Element Methods*. 1996.

[18] R. Speck. "Generalized Algebraic Kernels and Multipole Expansions for Massively Parallel Vortex Particle Methods". PhD thesis. Wuppertal University, 2011.

[19] D. Hilbert. "Ueber die stetige Abbildung einer Line auf ein Flächenstück". In: *Mathematische Annalen* 38 (3 1891), pp. 459–460.

[20] H. Sagan. *Space-Filling Curves*. 1st ed. Springer, Sept. 1994.

[21] G. M. Morton. "A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing". In: *IBM Germany Scientific Symposium Series*. 1966.

[22] B. Moon, H. v. Jagadish, C. Faloutsos, and J. H. Saltz. "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve". In: *IEEE Transactions on Knowlegde and Data Engineering* 13 (1 Jan. 2001), pp. 124–141.

[23] B. Assmann and P. Selke. *Technische Mechanik. 3. Kinematik und Kinetik*. Technische Mechanik / von Bruno Assmann. Oldenbourg, 2004.

[24] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration - Structure Preserving Algorithms for Ordinary Differential Equations*. 2. Springer Series in Computational Mathematics Vol. 31. Heidelberg: Springer, 2002.

[25] J. M. Dawson. "Particle Simulation of Plasmas". In: *Reviews of Modern Physics* 55.2 (Apr. 1983), pp. 403–447.

[26] R. W. Hockney. "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis". In: *Journal of the Association for Computing Machinery (ACM)* 12 (1 Jan. 1965), pp. 95–113.

[27] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301.

[28] J. Raeder. *Controlled nuclear fusion: fundamentals of its utilization for energy supply*. A Wiley-Interscience publication. Wiley, 1986.

[29] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. Bristol, PA, USA: Taylor & Francis, Inc., 1988.

[30] L. Greengard and V. Rokhlin. "A Fast Algorithm for Particle Simulations". In: *Journal of Computational Physics* 73 (2 Dec. 1987), pp. 325–348.

[31] B. Cipra. "The Best of the 20th Century: Editors Name Top 10 Algorithms". In: *SIAM News* 33 (4 May 2000), p. 1.

[32] F. H Harlow. *A Machine Calculation Method for Hydrodynamic Problems*. report. Los Alamos Scientific Laboratory, Nov. 1955.

[33] O. Bücker. "Parallelisierung der Nahfeldberechnung in der schnellen Multipolmethode für stark inhomogene Teilchensysteme". MA thesis. Jülich: FH Aachen, Standort Jülich, July 2009.

[34] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[35] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[36] K. A. Robbins and S. Robbins. *The Cray X-MP/model 24: a Case Study in Pipelined Architecture and Vector Processing*. New York, NY, USA: Springer-Verlag New York, Inc., 1989.

[37] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. "GPU Cluster for High Performance Computing". In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. SC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 47–.

[38] Message Passing Interface Forum. *MPI: a Message-Passing Interface standard, Version 2.2*. High Performance Computing Center, 2009.

[39] J. Nieplocha and B. Carpenter. "ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems". In: *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. London, UK: Springer-Verlag, 1999, pp. 533–546.

[40] M. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Trans. Comput.* C-21 (1972), pp. 948+.

[41] G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485.

[42] J. L. Gustafson. "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31 (5 May 1988), pp. 532–533.

[43] Y. Shi. *Reevaluating Amdahl's Law and Gustafson's Law*. Oct. 1996.

[44] *TOP 500 - November 2010*. TOP 500. visited: 24.01.2011, 15:01. URL: http://www.top500.org/list/2010/11/100.

[45] *GREEN 500 - November 2010*. GREEN 500. visited: 24.01.2011, 18:21. URL: http://www.green500.org/lists/2010/11/top/list.php?from=1&to=100.

[46] *JUROPA/HPC-FF - An Overview*. Forschungszentrum Jülich GmbH. visited: 24.01.2011, 23:14. URL: http://www.fz-juelich.de/jsc/files/docs/vortraege/supercomputer/SC-JUROPA-Introduction.pdf.

[47] S. Graf. "Entwicklung eines Werkzeugs zur Analyse des OS-Jitter Effekts auf High-Performance Cluster-Systemen". MA thesis. Jülich: FH Aachen, Standort Jülich, Jan. 2009.

[48] *JUGENE - An Overview*. Forschungszentrum Jülich GmbH. visited: 24.01.2011, 23:13. URL: http://www.fz-juelich.de/jsc/files/docs/vortraege/supercomputer/SC-JUGENE-Introduction.pdf.

[49] S. Pfalzner and P. Gibbon. "Direct Calculation of Inverse-Bremsstrahlung Absorption in Strongly Coupled, Nonlinearly Driven Laser Plasmas". In: *Physics Review E* 57.4 (Apr. 1998), pp. 4698–4705.

[50] P. Gibbon, F. N. Beg, E. L. Clark, R. G. Evans, and M. Zepf. "Tree-Code Simulations of Proton Acceleration from Laser-Irradiated Wire Targets". In: *Physical Review E* 11.4 (Aug. 2004), pp. 4032–4040.

[51] P. Gibbon, G. Münster, and D. Wolf. "Resistively Enhanced Proton Accelaration via High-Intensity Laser Interactions with Cold Foil Targets". In: *Physics of Plasmas* 72 (2005).

[52] D. Madlener. *Numerische Simulation von protostellaren Scheiben mit Treecodes and Smothed Particle Hydrodynamics*. Diploma thesis. 1. Physikalisches Institut der Universität Köln, Mar. 2008.

[53] A. Breslau. "Developement of a Parallel, Tree-Based Neighbour-Search Algorithm". In: Proceedings of the JSC Guest Student Programme on Scientific Computing. 2010.

[54] U. Becciani, V. Antonuccio-Delogu, and M. Gambera. "A Modified Parallel Tree Code for N-Body Simulation of the Large-Scale Structure of the Universe". In: *Journal of Compututational Physics* 163 (Sept. 2000), pp. 118–132.

[55] J. Dubinsky. "A Parallel Tree Code". In: *New Astronomy* 1 (1996), pp. 133–147.

[56] H. Yahagi, M. Mori, and Y. Yoshii. "The Forest Method as a New Parallel Tree Method with the Sectional Voronoi Tessellation". In: *The Astrophysical Journal Supplement Series* 124.1 (1999), p. 1.

[57] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. "On the Versatility of Parallel Sorting by Regular Sampling". In: *Parallel Comput.* 19 (10 Oct. 1993), pp. 1079–1103.

[58] P. Gibbon, R. Speck, L. Arnold, M. Winkel, and H. Hübner. "Parallel Tree Codes". In: *Proceedings der WE-Heraeus Summer School 2010, Fast Methods for Long-Range Interactions in Complex Systems*. 2011, pp. 65–84.

[59] A. M. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1988.

[60] M. Hofmann. *Paralleles Sortieren am Beispiel der schnellen Multipolmethode*. Tech. rep. JUEL–4211. Forschungszentrum Jülich, 2006.

[61] P. Gibbon, M. Hofmann, G. Rünger, and R. Speck. "Parallel Sorting Algorithms for Optimizing Particle Simulations". In: *2010 IEEE International Conference on Cluster Computing, Workshops and Posters (CLUSTER WORKSHOPS)*. IEEE, 2010, pp. 1–8.

[62] H. Dachsel, M. Hofmann, and G. Rünger. "Library Support for Parallel Sorting in Scientific Computations". In: *Proc. of the 13th International Euro-Par Conference*. Vol. 4641. LNCS. Springer, 2007, pp. 695–704.

[63] M. Winkel, R. Speck, H. Hübner, L. Arnold, P. Gibbon, and R. Krause. "A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations". In: *Computer Physics Communication* (submitted).

[64] T. Hamada and K. Nitadori. "190 TFlops Astrophysical N-Nody Simulation on a Cluster of GPUs". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9.

[65] G. Peano. "Sur une courbe, qui remplit toute une aire plane." In: *Mathematische Annalen* 36 (1890), pp. 157–160.

[66] E. H. Moore. "On Certain Crinkly Curves." In: *Transaction of the American Mathematical Society* 1 (1900), pp. 72–90.

[67] H. Lebesque. "Lecons sur l'integration et la recherche des fonctions primitives." In: (1904), pp. 44–45.

[68] K. Abend, T. Harley, and L. Kanal. "Classification of Binary Random Patterns". In: *IEEE Transactions on Information Theory* 11 (1965), pp. 538–544.

[69] J. K. Lawder and P. J. H. King. "Using Space-Filling Curves for Multi-dimensional Indexing". In: *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases*. BNCOD 17. London, UK: Springer-Verlag, 2000, pp. 20–35.

[70] C. Blatter. *Analysis 3*. Vol. 1. Springer, 1974, pp. 41–43.

[71] C. Jordan. “Remarques sur les integrales definies.” In: *Journal de Mathematiques Pures et Appliquees* 8 (1892), pp. 69–99.

[72] W. Sierpinsky. “Sur une nouvelle courbe continue qui remplit toute une air plane.” In: (1912), pp. 462–478.

[73] R. Zhang and C.-T. Zhang. “Identification of Replication Origins in Archaeal Genomes Based on The Z-Curve Method.” In: *Archaea* 1.5 (2005), pp. 335–46.

[74] A. R. Butz. “Alternative Algorithm for Hilbert’s Space-Filling Curve”. In: *IEEE Transactions on Computers* 20 (Apr. 1971), pp. 424–426.

[75] S. I. Kamata, R. O. Eason, and Y. Bandou. “A New Algorithm for N-Dimensional Hilbert Scanning”. In: *IEEE Transactions on Image Processing* 8.7 (1999), pp. 964–73.

[76] L. Chenyang and F. Yucai. “Algorithm for Analyzing N-Dimensional Hilbert Curve”. In: *International Conference on Web-Age Information Management, Dalian, China*. 2005, pp. 657–662.

[77] L. Chenyang, Z. Hong, and W. Nengchao. “Fast N-Dimensional Hilbert Mapping Algorithm”. In: *Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 507–513.

[78] X. Liu. “Four Alternative Patterns of the Hilbert Curve”. In: *Applied Mathematics and Computation* 147.3 (2004), pp. 741 –752.

[79] C. Geile. *Skalierbare Berechnung von Interprozess-Leistungsmetriken und deren Visualisierung*. Technical Report. Forschungszentrum Jülich GmbH, Dec. 2007.

[80] Homepage of VAMPIR. URL: http://www.vampir.eu/.

[81] Homepage of SCALASCA. URL: www.scalasca.org/.

Jül-4339
Juli 2011
ISSN 0944-2952